public static partial class Unsafe

{

// Casting/reinterpretation

public unsafe static void\* AsPointer<T>(ref T value)

public unsafe static ref T AsRef<T>(void\* source)

public static ref TTo As<TFrom, TTo>(ref TFrom source)

// Pointer arithmetic

public static ref T Add<T>(ref T source, int elementOffset)

public static ref T Subtract<T>(ref T source, int elementOffset)

// Informative methods

public static int SizeOf<T>()

public static System.IntPtr ByteOffset<T>(ref T origin, ref T target)

public static bool IsAddressGreaterThan<T>(ref T left, ref T right)

public static bool IsAddressLessThan<T>(ref T left, ref T right)

public static bool AreSame<T>(ref T left, ref T right)

// Memory access methods

public unsafe static T Read<T>(void\* source)

public unsafe static void Write<T>(void\* destination, T value)

public unsafe static void Copy<T>(void\* destination, ref T source)

// Block-based memory access

public static void CopyBlock(ref byte destination, ref byte source, uint byteCount)

public unsafe static void InitBlock(void\* startAddress, byte value, uint byteCount)

}

***Listing 14-38***Unsafe class API - some overloads removed for brevity, methods are reordered into feature-like groups, comments are my own

It is clear that Unsafe is not a general-purpose class. It can be used in only very specific, well-controlled places where the programmer really knows what it wants to do and considered all uncommon, boundary cases. Do not treat this class as a helper to overcome strange type-safety problems, for example, to break a type hierarchy in object-oriented programming!

Let’s look at few examples. First of all, we have already seen important Unsafe class usage in Listings [14-18](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC18), [14-20](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC20), [14-23](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC23), and [14-24](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC24) where casting and pointer arithmetic were used to implement Span<T>.

Casting is a powerful tool though. For example, we can cast one managed type to another, completely unrelated type (see Listing [14-39](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC39)). Memory of source instance is reinterpreted with respect to the field’s layout of the target instance. In our simple example we are just reinterpreting two successive integers as long, which may even make some sense. Please note that even such low-level pointers operations are used, DangerousPlays method is not marked as unsafe because Unsafe class wraps everything inside.

public class SomeClass

{

public int Field1;

public int Field2;

}

public class SomeOtherClass

{

public long Field;

}

public void DangerousPlays(SomeClass obj)

{

ref SomeOtherClass target = ref Unsafe.As<SomeClass, SomeOtherClass>(ref obj);

Console.WriteLine(target.Field);

}

***Listing 14-39***Dangerous but working code - casting with Unsafe.As

Such powerful casting is used, for example, to break mutability rules and allows them to cast between Memory<T> and ReadOnlyMemory<T> in both directions. This of course requires that both types have the same memory layout.

Casting is, for example, intensively used in BitConverter static class to convert from byte arrays back and forth to various types (see Listing [14-40](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC40)).

public static byte[] GetBytes(double value)

{

byte[] bytes = new byte[sizeof(double)];

Unsafe.As<byte, double>(ref bytes[0]) = value;

return bytes;

}

***Listing 14-40***Example of Unsafe usage in BitConverter class

While using all that memory reinterpretation, imagine primitive types reinterpreted into references or the other way around! Obviously, this is extremely dangerous and most probably will lead to the whole runtime crash. As an illustration, see Listing [14-41](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC41) as an example of such careless casting. VeryDangerous method will throw AccessViolationException (unless we are so unusual lucky that the value of Long1 had the value of the valid string).

public struct UnmanagedStruct

{

public long Long1;

public long Long2;

}

public struct ManagedStruct

{

public string String;

public long Long2;

}

public void VeryDangerous(ref UnmanagedStruct data)

{

ref ManagedStruct target = ref Unsafe.As<UnmanagedStruct, ManagedStruct>(ref data);

Console.WriteLine(target.String); // Value of Long1 is now treated as string reference!

}

***Listing 14-41***Very dangerous code - casting with Unsafe.As

Pointer arithmetic is the other popular usage of Unsafe. As a good example, consider the may serve Array.Reverse static method implementation (see Listing [14-42](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC42)). This is nothing else than a reincarnation of regular C or C++-like code manipulating pointers to reverse an array in place.

public static void Reverse<T>(T[] array, int index, int length)

{

...

ref T first = ref Unsafe.Add(ref Unsafe.As<byte, T>(ref array.GetRawSzArrayData()), index);

ref T last = ref Unsafe.Add(ref Unsafe.Add(ref first, length), -1);

do

{

T temp = first;

first = last;

last = temp;

first = ref Unsafe.Add(ref first, 1);

last = ref Unsafe.Add(ref last, -1);

} while (Unsafe.IsAddressLessThan(ref first, ref last));

}

***Listing 14-42***Example of Unsafe usage in Array.Reverse static method

Because many Span<T>, Memory<T>, and Unsafe usages require the same patterns, the MemoryMarshal helper class was introduced with many static methods. To name only a few of them:

* AsBytes - converts any Span<T> of primitive type (struct) to Span<byte>,
* Cast - converts between two Span<T> of primitive types (structs),
* TryGetArray, TryGetMemoryManager, TryGetString - tries to convert from given Memory<T> (or ReadOnlyMemory<T>) to a specific type,
* GetReference - to ref return underlying Span<T> or ReadOnlySpan<T> object.

With the MemoryMarshal class we can even more easily do “magic” things. For example, we can take a part of some struct and reinterpret it as another struct, all without any copying (see Listing [14-43](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC43)).

public struct SmallStruct

{

public byte B1;

public byte B2;

public byte B3;

public byte B4;

public byte B5;

public byte B6;

public byte B7;

public byte B8;

}

public unsafe void Reinterpretation(ref UnmanagedStruct data)

{

var span = new Span<UnmanagedStruct>(Unsafe.AsPointer(ref data), 1);

ref var part = ref MemoryMarshal

// cast from Span<byte> to Span<SmallStruct>

.Cast<byte, SmallStruct>(

// cast from Span<UnmanagedStruct> to Span<byte>

MemoryMarshal.AsBytes(span)

// slice accordingly and access first element

.Slice(0, 8))[0];

Console.WriteLine(part.B1); // Get the first byte

}

***Listing 14-43***Example of MemoryMarshal usage

One may wonder where all that “magic” may be useful for him. Does a regular .NET developer need Unsafe at all? To be honest, mostly not. I imagine Unsafe usage only in low-level operating libraries code - serialization, binary logging, network communication, and so on, so forth. For example, popular jemalloc.NET library uses it to provide strong typing over underlying unmanaged memory (see Listing [14-44](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC44)).

[MethodImpl(MethodImplOptions.AggressiveInlining)]

public unsafe ref C Read<C>(int index) where C : struct

{

return ref Unsafe.AsRef<C>(PtrTo(index));

}

***Listing 14-44***Example of Unsafe usage in jemalloc.NET - FixedBuffer.Read method

jemalloc.NET is a great .NET library written by Allister Beharry and hosted on GitHub (<https://github.com/allisterb/jemalloc.NET>). As the author says, it is a wrapper “over the jemalloc native memory allocator and provides .NET applications with efficient data structures backed by native memory for large scale in-memory computation scenarios.” jemalloc is indeed a popular and efficient malloc replacement. Feel free to read about its internal implementation at <http://jemalloc.net/>and also feel invited to experiment with jemalloc.NET. Due to the book=size limitations, not without regret, I have to skip a description of this library.

Speaking of unmanaged memory wrappers, there is also ongoing work on the Microsoft side - project Snowflake. Currently its status is a little frozen but expect open sourcing it sooner or later. You can read about it on <https://www.microsoft.com/en-us/research/publication/project-snowflake-non-blocking-safe-manual-memory-management-net/>site.

### **UNSAFE INTERNALS**

In fact, what Unsafe class really does is wrap various IL-based possibilities that are otherwise not possible to express in C# - because IL type control is less strict than that incurred by C# compiler. CIL implementation of most Unsafe methods are really trivial (see Listing [14-45](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC45)).

.method public hidebysig static !!TTo& As<TFrom, TTo> (!!TFrom& source) cil managed

{

IL\_0000: ldarg.0

IL\_0001: ret

}

.method public hidebysig static !!T& Add<T> (!!T& source, int32 elementOffset) cil managed

{

IL\_0000: ldarg.0

IL\_0001: ldarg.1

IL\_0002: sizeof !!T

IL\_0008: conv.i

IL\_0009: mul

IL\_000A: add

IL\_000B: ret

}

***Listing 14-45***Example of Unsafe method implementation (in Common Intermediate Language)

There is no magic underneath Unsafe though. What makes it really useful is exposing all those operations, most often consumable even in safe code.

## **Data-Oriented Design**

The discrepancy between CPU performance and memory access times are constantly growing. We have discussed it already in Chapter [2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml) quite comprehensively - how CPU and memory cooperation are organized into hierarchical cache and how significantly its organization into cache lines and memory internal implementation influences performance of code we write, preferring sequential data access with strong temporal and spatial locality.

Such a low-level view of memory access is not crucial during everyday development of business-driven, regular web, or desktop applications. Those milliseconds of better or worse performance aren’t simply noticeable in small volume of processed data, processed HTTP requests, or handled UI interactions. Readability, extensibility, and expressiveness of the source code, as well as the ability to write, deliver, and extend software fast, are the most important factors when designing such applications. Object-oriented programming, with all its design patterns and SOLID principles, are an exact incarnation of such approach.

However, there is a narrow category of applications that can benefit from breaking this universal convention. These are applications that have to process significant amounts of data in the most efficient way and shortest possible time. Where every millisecond counts. To name a few such examples:

* financial software - especially real-time trading and any analytical decisions may require as fast-as-possible answer based on significant amount of various data.
* Big Data - although in general we may associate it more with batch, slow processing, every millisecond per data processing operation can sum up to a difference of hours or days of overall processing. And still, there are applications where fast answer does really count - like search engines.
* games - in a world where FPS (Frames per seconds) decides on game reception and limits possible graphics quality, every millisecond matters.
* machine learning - there is always not enough processing power to execute various, complicated algorithms used in gaining popularity ML.

Please note that although, at first glance, many of those applications could be CPU-bound (i.e., contains complex algorithms to be executed), because of the above-mentioned discrepancy, it may be memory access that introduces a performance bottleneck. Another, not-yet mentioned aspect is parallel processing of the data, to benefit from multiple logical cores installed on our personal or server computers.

This leads us to *data-oriented design* of software - concentrated around designing data representation and architecture that lead to the most efficient memory access. It almost certainly stays in contradiction to the object-oriented design, because techniques like encapsulation or polymorphism are interfering with achieving effective memory utilization.

What data-oriented design is trying to leverage is:

* designing types and data in a way that lead to a sequential memory access wherever possible, taking into consideration cache-line limits (to pack together most frequent used data) and hierarchical cache nature (to keep as much in higher caches as possible).
* designing types and data, as well as algorithms using them, in a way that leads to easy parallelization without costly synchronization.

I would further split data-oriented design into two more categories:

* *tactical data-oriented design* - concentrates on “local” data structures, like most efficient field’s layout or accessing data in correct order. Such design is local enough to be incorporated quite easily into already existing object-oriented applications.
* *strategic data-oriented design* - concentrates on high-level view of the application, from architecture perspective. It mostly requires mindset shift from object-oriented structures into more data-oriented ones.

In the two subsequent sections we will look deeper at both mentioned aspects of such design.

### **TACTICAL DESIGN**

This book is basically steeped with the spirit of tactical data-oriented design since Chapter [2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml), where we have learned how important cache utilization is - and summarized in *Rule 2 - Random access should be avoided* and *Rule 3 - Improve spatial and temporal data locality*.

Several patterns constitute such tactical design. Let’s summarize them here a little, with appropriate references from the rest of the book and additional examples.

#### ***Design Types to Fit as Much Relevant Data as Possible in the First Cache Line***

We have seen this rule in action when considering the automatic memory layout of managed types - references all laid at the beginning of the object to make them accessible for the GC within already accessed cache line containing MethodTable pointer. This is optimization done by CLR but we should be aware of it.

Such automatic layout may be, or may not be, a desired one when considering the most commonly accessed data. Imagine the class from Listing [14-46](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC46). Obviously, the object-oriented programmer will be quite happy with such design[4](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn4) - everything is encapsulated within a single object and only behavior (calculating scoring) is publicly exposed.

class Customer

{

private double Earnings;

// ... some other fields ...

private DateTime DateOfBirth;

// ... some other fields ...

private bool IsSmoking;

// ... some other fields ...

private double Scoring;

// ... some other fields ...

private HealthData Health;

private AuxiliaryData Auxiliary;

public void UpdateScoring()

{

this.Scoring = this.Earnings \* (this.IsSmoking ? 0.8 : 1.0) \*

ProcessAge(this.DateOfBirth);

}

private double ProcessAge(DateTime dateOfBirth) => 1.0;

}

***Listing 14-46***Example class used to illustrate cache line utilization

Such a programmer will not be completely interested in the resulting automatic layout of the Customer object. On the other hand, imagine that we use Customer class massively, mainly calling UpdateScoring on millions of such instances per second. As UpdateScoring method uses Scoring, Earning, IsSmoking, and DateOfBirth fields, they should be laid out within the range of the first cache line (the one accessed always when Customer instance is used). LayoutKind.Automatic , default one for classes, obviously doesn’t care about that. It will put, probably very rarely used, HealthData and AuxiliaryData references at the beginning of the object while the rest will be laid out according to alignment requirements (as explained in Object/struct layout section in the previous chapter).

The solution should be already known to us - we must change Customer into unmanaged struct that may use sequential layout (see Listing [14-47](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC47)). It may be done by:

* changing HealthData and AuxiliaryData into value-type identifiers, to get rid of references - this helps not only in changing such type into unmanaged type, it will also relieve the GC from marking overhead (as each Customer instance will not be a root of two additional objects to be scanned).
* changing DateTime to other type as its automatic layout triggers automatic layout of the whole struct, as described in Chapter [13](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_13_Chapter.xhtml).

Then we may use LayoutKind.Sequential, carefully designing the layout of the fields on our own (considering padding introduced due to the alignment, but probably we can sell some space in favor of the speed). Thus, the four most commonly used fields should be placed at the beginning.

[StructLayout(LayoutKind.Sequential)]

struct CustomerValue

{

public double Earnings;

public double Scoring;

public long DateOfBirthInTicks;

public bool IsSmoking;

// ... some other fields ...

public int HealthDataId;

public int AuxiliaryDataId;

}

***Listing 14-47***Struct with layout considering cache-line utilization

However, not always, we must use sequential layout to achieve good spatial locality. Sometimes it is just enough to make sure that data locality of primitive types is simply taken care of (in other words, it is assured that commonly accessed fields are laid out next to each other).

FrugalObjectList<T> and FrugalStructList<T> are an example of very interesting internal collections used inside Windows Presentation Library. Their internal storage is an instance of one of the following, specific collections: SingleItemList<T>, ThreeItemList<T>, SixItemList<T>, and ArrayItemList<T>. While adding or removing elements, such storage is converted between those types (while the last one handles storage of seven or more items). What does it give in return? A very concise, trivial, and mostly switch-based implementations of methods like IndexOf, SetAt or EntryAt, used by indexer, for scenarios with less than seven elements (see Listing [14-48](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC48), showing fragments of ThreeItemList<T>). So while getting rid of generic array overhead (bounds checking, to name one), such an approach still provides good spatial locality because of three or six fields laid out next to each other.

/// <summary>

/// A simple class to handle a list with 3 items. Perf analysis showed

/// that this yielded better memory locality and perf than an object and an array.

/// </summary>

internal sealed class ThreeItemList<T> : FrugalListBase<T>

{

public override T EntryAt(int index)

{

switch (index)

{

case 0:

return \_entry0;

case 1:

return \_entry1;

case 2:

return \_entry2;

default:

throw new ArgumentOutOfRangeException("index");

}

}

private T \_entry0;

private T \_entry1;

private T \_entry2;

}

***Listing 14-48***Fragments of ThreeItemList<T> class (one of storages used by FrugalObjectList<T> and FrugalStructList<T> types)

As those types comment says: “Performance measurements show that Avalon[5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn5) has many lists that contain a limited number of entries, and frequently zero or a single entry. (...) Therefore these classes are structured to prefer a storage model that starts at zero, and employs a conservative growth strategy to minimize the steady state memory footprint. (...) The code is also structured to perform well from a CPU standpoint. Perf analysis shows that the reduced number of processor cache misses makes FrugalList faster than ArrayList or List<T>, especially for lists of 6 or fewer items.”

#### ***Design Data to Fit into Higher Cache Levels***

Overhead of various cache levels has been already illustrated in Listing [2-5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml#PC5) and corresponding Figure [2-11](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml#Fig11) in Chapter [2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml). You should be always aware how big your data is and how it relates to the typical CPU cache sizes.

#### ***Design Data That Allows Easy Parallelization***

Topic of parallel processing goes out of the scope of this book. However, good data layout and algorithm design may allow some parts of the data to be processed in parallel - whether it be multiple cores and/or SIMD instructions. Remember still about the false-sharing caveat illustrated in Listing [2-6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml#PC6) and corresponding benchmark in Table [2-3](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml#Tab3).

#### ***Avoid Non-sequential, Especially Random Memory Access***

This rule has been explained in Chapter [2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml), starting from explaining how DRAM works and why sequential access is preferred. A simple example of accessing a two-dimensional array by rows versus by columns was shown in Listing [2-1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml#PC1) and corresponding benchmark in Table [2-1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml#Tab1), showing several times slower access due to a lot of cache miss.

Accessing the sequentially contiguous memory region of T[] is a preferred way over other collections, especially if T is a struct (recall Figure [4-22](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_4_Chapter.xhtml#Fig22) from the chapter comparing data locality of arrays). We will make use of this design rule when describing strategic patterns.

### **STRATEGIC DESIGN**

Strategic design pushes forward data-oriented design, leaving far behind typical object-oriented design practices. Code it produces may be surprising to developers used to OOP but become more and more justified if you think about it deeply. Therefore, unlike tactical design, strategic design requires a significant mind-shift of the programmer. Let’s now look at some of the most popular techniques.

#### ***Moving from Array-of-Structures to Structure-of-Arrays***

In object-oriented programming, data is encapsulated. Objects and methods are representing well-crafted, single responsibility behaviors. For example, we can imagine that Customer instances from Listing [14-46](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC46) are kept by separate “container.” Its UpdateScorings method enumerates all customer instances and ask them to update their scoring (see Listing [14-49](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC49)). This is a plain and simple code that every developer using OOP would understand.

class CustomerRepository

{

List<Customer> customers = new List<Customer>();

public void UpdateScorings()

{

foreach (var customer in customers)

{

customer.UpdateScoring();

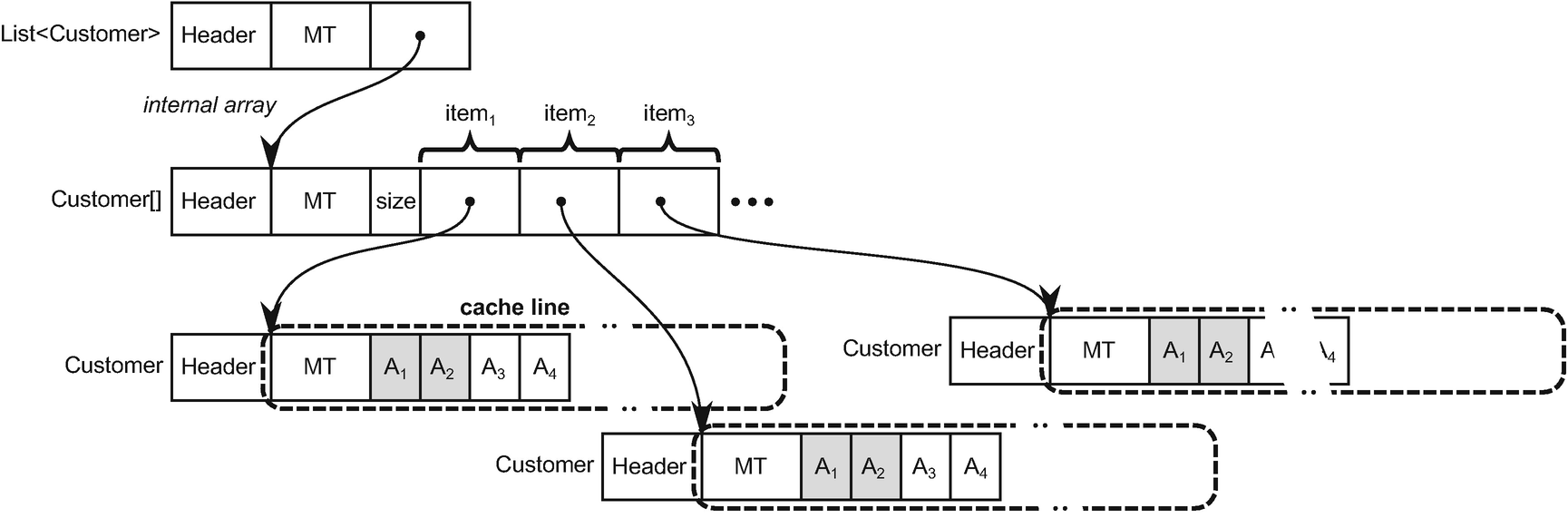
}

}

}

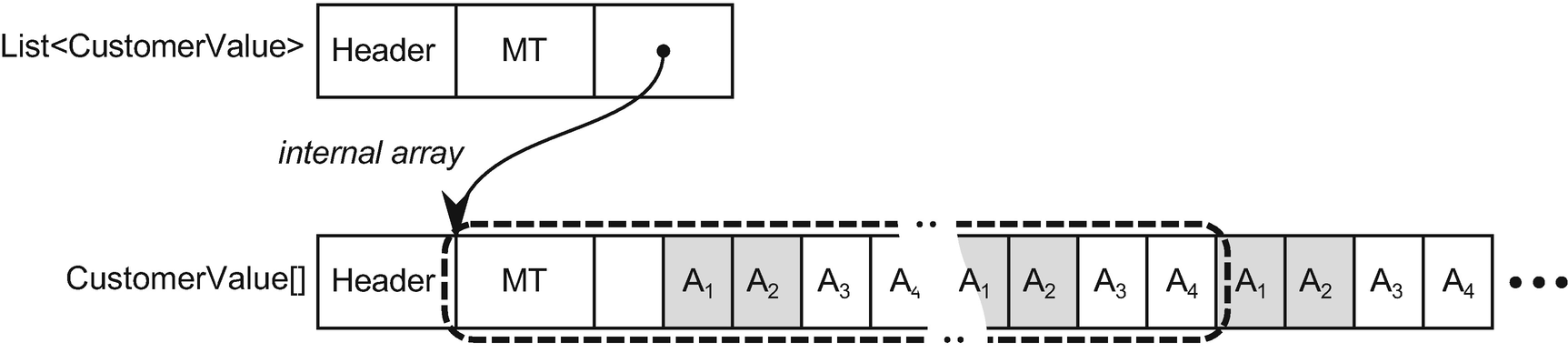
***Listing 14-49***Repository of customers from Listing [14-46](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC46)

Such code introduces a lot of cache-line misses - Customer instances may be scattered all around the GC Heap as there is no guarantee that they are allocated next to each other (see Figure [14-1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fig1)). Although, as we know, compacting GC eventually may lead to good data locality of objects allocated around the same time. Additionally, a bump-a-pointer allocator may allocate them next to each other in the first place. But those are assumptions, not guarantees. For example, because filled allocation context will be changed into a new one, possibly all around the ephemeral segment, even two successive Customer allocations may land in two completely different places. As a result, we must assume that in case of array of reference types, each cache line consists of only a small part of interesting data and a lot of surrounding garbage.



***Figure 14-1***Poor data locality of reference type array leads to many cache lines reading a lot of unnecessary data (necessary data is grayed)

We know that array of structs provides much better data locality so CustomerRepository instead of Customer instances could store a list of boxed CustomerValue instances, defined in Listing [14-47](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC47) (see Figure [14-2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fig2)). Successive reading of List’s underlying array utilizes cache lines much better as CPU’s prefetcher will easily recognize such pattern and will prefetch data in advance. There is also much less memory garbage read into each cache line - it consists only of other, currently not needed fields of CustomerValue instance.



***Figure 14-2***Much better data locality of value-type array leads to cache lines reading a lot less of unnecessary data (necessary data is grayed)

However, reading those unnecessary data (fields) may be still too costly in performance-critical scenarios. At this moment it’s high time we left well-known OOP paradigms and changed things all around. In data-oriented design, the most important are not objects and behaviors they encapsulate, but the data itself. In our case the data consist of a few important attributes of customer (both as input and output).

The first approach would be to split customer data into two separate arrays of value types - one containing “hot data” used in scoring algorithm, the second with the rest, less relevant fields.

But we may go even further. So instead of gathering code around the customer, we may organize them around the data itself - by exposing each relevant data with a separate array (see Listing [14-50](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC50)). Such approach is one of the most popular in data-oriented design, often referred to as changing the layout from *AoS* (*array-of-structures*) to *SoA* (*structure-of-arrays*).

class CustomerRepository

{

int NumberOfCustomers;

double[] Scoring;

double[] Earnings;

DateTime[] DateOfBirth;

bool[] IsSmoking;

// ...

public void UpdateScorings()

{

for (int i = 0; i < NumberOfCustomers; ++i)

{

Scoring[i] = Earnings[i] \* (IsSmoking[i] ? 0.8 : 1.0) \* ProcessAge(DateOfBirth[i]);

}

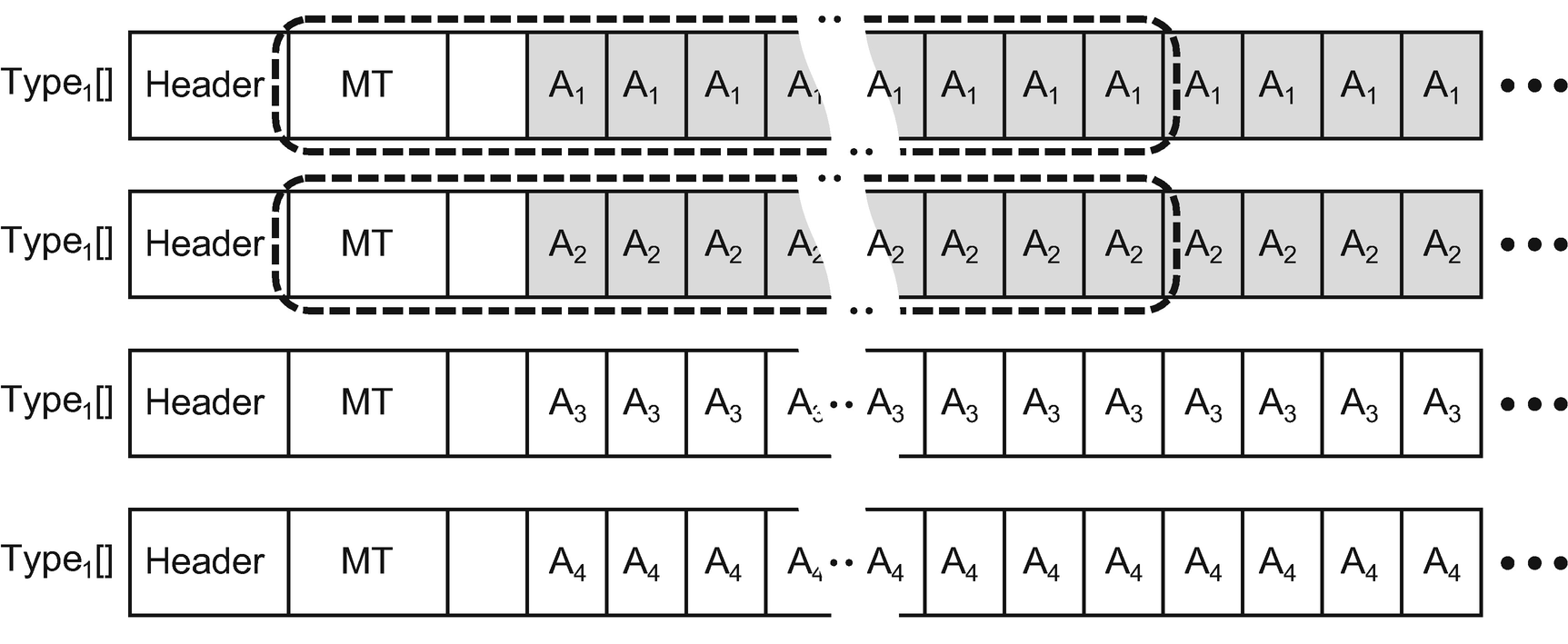
}

...

}

***Listing 14-50***Structure-of-arrays data organization example

By directly exposing the data, there is in fact no “customer” entity in such approach. “Customer” is just a bunch of data under a specific index in respective arrays. Those arrays are densely packed with relevant data, accessed sequentially by our hot-path algorithm. Cache-line utilization is optimal (see Figure [14-3](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fig3)). CPU can detect multiple sequential reads simultaneously so prefetcher will be used in each array access.



***Figure 14-3***Optimal data locality in structure-of-arrays approach (necessary data is grayed)

As an additional advantage, the struct-of-arrays approach provides nice flexibility. If we introduce other high-performance algorithm use at other time, using different fields, such data organization will be beneficial also.

In a similar way we may flatten hierarchical (tree) data. Typically, each node would be storing a list of its children. Obviously, traversal of such tree may be quite costly due to the cache misses while accessing heap-allocated node instances scattered all around the GC Heap.

Let’s use a trivial tree example from Listing [14-51](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC51), which implements also simple, exemplary algorithm - Process method changes value of each node into a sum of values from its ancestors.[6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn6)

public class Node

{

public int Value { get; set; }

public List<Node> Children = new List<Node>();

public Node(int value) => Value = value;

public void AddChild(Node child) => Children.Add(child);

public void Process()

{

InternalProcess(null);

}

private void InternalProcess(Node parent)

{

if (parent != null)

this.Value = this.Value + parent.Value; // Imagine more complex processing here

foreach (var child in Children)

{

child.InternalProcess(this);

}

}

}

***Listing 14-51***Simple tree with nodes implementation

However, such tree may be described quite oppositely by a flat array of nodes - each element being a node, storing a reference (or better, an index) of its parent. Such approach most probably will require preprocessing of an initial, more natural, object-oriented tree into such an array. Processing of such tree may be then linear, if it was appropriately flattened (see Listing [14-52](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC52)).

public class Tree

{

public struct ValueNode

{

public int Value;

public int Parent;

}

private ValueNode[] nodes;

private static Tree PrecalculateFromRoot(OOP.Node root)

{

// Flatten tree navigating it in pre-order depth-first manner...

}

public void Process()

{

for (int i = 1; i < nodes.Length; ++i)

{

ref var node = ref nodes[i];

node.Value = node.Value + nodes[node.Parent].Value;

}

}

}

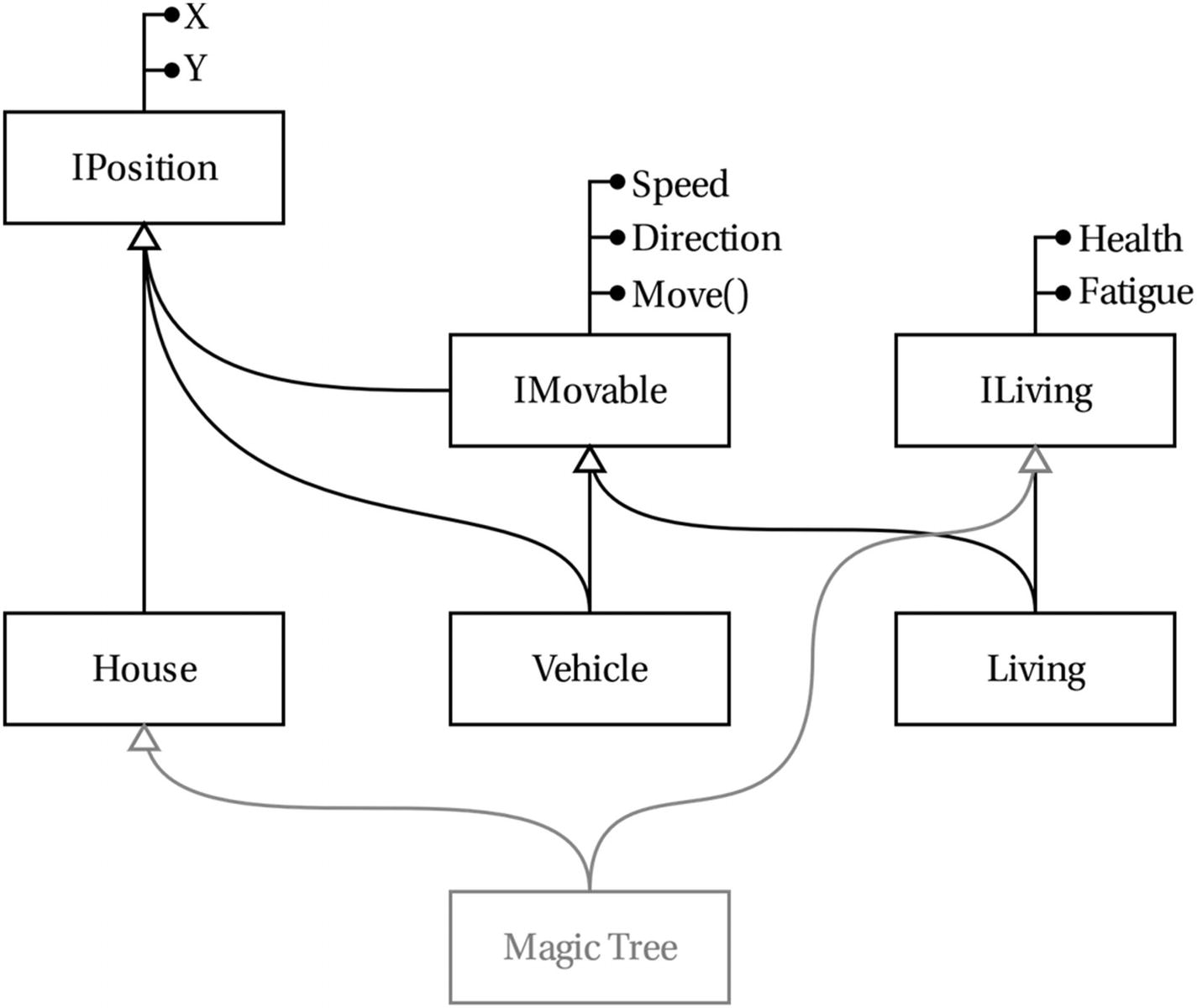
***Listing 14-52***Example of flattened tree, represented as array of value-type nodes

Please be careful when designing tree flattening. The particular example from Listing [14-52](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC52) works because the used processing algorithm (value adding inside Process method) depends only on parent values so it is perfectly fine to use a pre-order depth-first traversal. After, such flattening elements in the nodes array are always located after the already processed parent. If our algorithm depended on children (like a node value being a sum of all its descendants), post-order depth-first traversal should be used, which guarantees that each element of the flattened array is after all its children.

#### ***Entity Component System***

In object-oriented programming, inheritance and encapsulation are one of the core features. In complex applications, inheritance tree may be quite complicated, with many objects sharing some part of possible behaviors. Games are perfect example of scenario where there are dozens of various types of differently behaving entities - for example, tanks being armored vehicles while trucks being vehicles not armored but they are containers. Or a regular solider being only movable and having attributes like health, but is not always armored. A sample inheritance tree to illustrate that is presented in Figure [14-4](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fig4).

In the broader context of software development, such inheritance tree may be cumbersome because adding a new kind of entity that shares only part of possible behaviors is not trivial - it must be added, overriding appropriate methods to include new behavior, and so on, and so forth (like adding MagicTree class in Figure [4-4](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_4_Chapter.xhtml#Fig4), which is both “positionable” and is a living - but is not movable).



***Figure 14-4***Example of inheritance tree representing some game objects

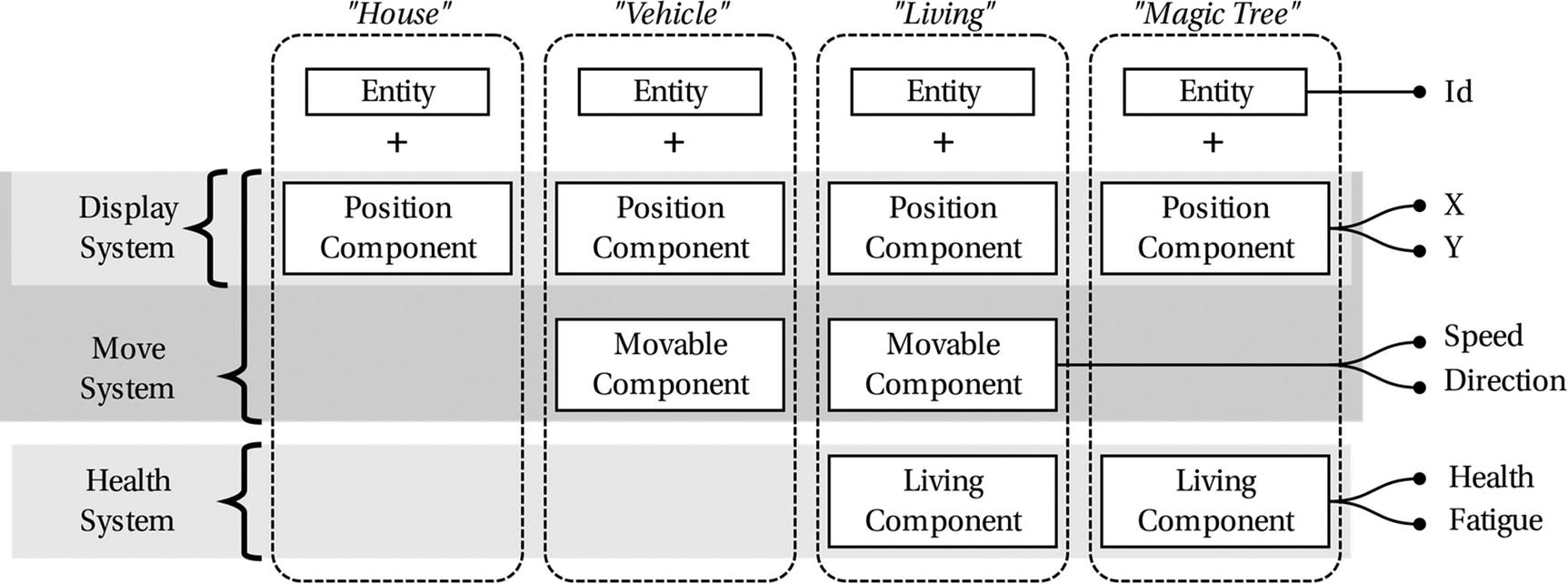
In our data-oriented context, caveats of such approach should be immediately visible - data is spread all around such tree hierarchy. It is perfectly OK in regular OOP, where there are few business objects cooperating with each other. But it becomes bottleneck if we have to process thousands or millions similar entities, let’s say - vehicles, to update their position.

We could use the structure-of-arrays approach to keep separate list of structs representing houses, vehicles, livings, and so on, so forth. This however is not very practical, and still many algorithms may need to access various set of properties contained in those lists (breaking good data locality benefits).

The solution to this problem is proposed into form of so-called *Entity Component System* that, simply speaking, prefers composition over inheritance. As we will soon see, one of its foundations is good data locality consistent with the idea of structure-of-arrays.

In Entity Component System, there are no types representing house, vehicle, or any other living. Entities are being composed by dynamically adding and removing components, representing capabilities. Such entities are then processed by various systems, representing required logic. In other words, the three main building blocks in ECS are (see Figure [14-5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fig5)):

* *Entity* - is a simple object with an identity but does not contain any data or logic. By adding or removing specific Components to it, we define capabilities of such Entity. So, for example, when we need something like a vehicle in a game, we create an entity and assign appropriate components to it (Position and Movable component in our simplified example).
* *Component* - simple object only consisting of data but no logic. Those data are needed to represent current state of the capability represented by such component (so position in Position component or speed in Movable component).
* *System* - is where the logic of specific capabilities or features lives. Systems operate on filtered list of entities, one by one. For example, Move System will filter all entities to those that have Position and Movable components assigned (and its logic knows how to transform/process properties of those components).



***Figure 14-5***Overview of Entity Component System

In a main loop of a game, each system executes one after another. I hope it is already visible where the power of such approach is lying. With such design, data of each component are kept sequentially and separately, incorporating structure-of-arrays approach. For example, when Display System iterates through entities, it in fact needs to iterate over sequential collection of Position Component data. Obviously, it requires a very efficient filtering technique of entities (or answering the question whether entity has given component attached). Those are, however, implementation details we will not touch here. Instead, let’s implement the simplest possible ECS we can imagine. Hopefully it will allow us to illustrate the whole concept better.

First of all, Entity may be really simple type containing only identifier (see Listing [14-53](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC53)). It is a readonly struct - to keep it densely in the array of entities and to avoid defensive copies when passing around as in arguments.

public readonly struct Entity

{

public readonly long Id;

public Entity(long id)

{

Id = id;

}

}

***Listing 14-53***Entity definition

Components are also only simple containers for data. Again, to make a dense array of component data, they are structs (see Listing [14-54](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC54)). They are mutable and thanks to ref returns, we will be able to return them from the corresponding storage for modification.

public struct PositionComponent

{

public double X;

public double Y;

}

public struct MovableComponent

{

public double Speed;

public double Direction;

}

public struct LivingComponent

{

public double Fatigue;

}

***Listing 14-54***Sample components definitions

To effectively store data of a given component in a data-oriented way, let’s introduce ComponentManager<T> class (see Listing [14-55](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC55)). Its main part is registeredComponents array of a given component type. Registering is as easy as filling the next free slot in the array (and for brevity I’ve skipped a problem of unregistering and resulting fragmentation). Checking whether given entity (identified by its Id) has component assigned is based on an additional dictionary - this is again by far the most efficient way but it was used for brevity (as well as ignoring any multithreading issues). Its ref returns an array element so no copying is involved.

public class ComponentManager<T>

{

private static T Nothing = default;

private static int registeredComponentsCount = 0;

private static T[] registeredComponents = ArrayPool<T>.Shared.Rent(128);

private static Dictionary<long, int> entityIdtoComponentIndex = new Dictionary<long, int>();

public static void Register(in Entity entity, in T initialValue)

{

registeredComponents[registeredComponentsCount] = initialValue;

entityIdtoComponentIndex.Add(entity.Id, registeredComponentsCount);

registeredComponentsCount++;

}

public static ref T TryGetRegistered(in Entity entity)

{

if (entityIdtoComponentIndex.TryGetValue(entity.Id, out int index))

{

//result = true;

return ref registeredComponents[index];

}

//result = false;

return ref Nothing;

}

}

***Listing 14-55***ComponentManager<T> class managing component data

Them we need an abstract representation of the system (see Listing [14-56](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC56)) and a manager that ties all this together (see Listing [14-57](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC57)).

public abstract class SystemBase

{

public abstract void Update(List<Entity> entities);

}

***Listing 14-56***Definition of simple abstract system base

public class Manager

{

private List<Entity> entities = new List<Entity>();

private List<SystemBase> systems = new List<SystemBase>();

public void RegisterSystem(SystemBase system)

{

systems.Add(system);

}

public Entity CreateEntity()

{

var entity = new Entity(entities.Count);

entities.Add(entity);

return entity;

}

public void Update()

{

foreach (var system in systems)

{

system.Update(entities);

}

}

}

***Listing 14-57***Manager storing list of entities and systems

Having all those bricks in place, it’s high time to write an example system. MoveSystem requires entities with both Position and Movable components, so its Update methods filters them appropriately (see Listing [14-58](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC58)). The requirement of very efficient entities filtering is clearly visible here. However, if managed properly, data components are accessed sequentially with a high probability, providing great data locality and prefetching possibility.

public class MoveSystem : SystemBase

{

public override void Update(List<Entity> entities)

{

foreach (var entity in entities)

{

bool hasPosition = false;

bool isMovable = false;

ref var position = ref ComponentManager<PositionComponent>.TryGetRegistered(in entity, out hasPosition);

ref var movable = ref ComponentManager<MovableComponent>.TryGetRegistered(in entity, out isMovable);

if (hasPosition && isMovable)

{

position.X += CalculateDX(movable.Speed, movable.Direction);

position.Y += CalculateDY(movable.Speed, movable.Direction);

}

}

}

}

***Listing 14-58***An example of Moving system

Please note that provided implementation is oversimplified in many places. As mentioned, it does not include any thread synchronization, and proposed entity-to-component management is also trivialized. Presenting here a full, even closely real-world implementation is by far behind such book capacity. In real-world libraries, like Entitas (<https://github.com/sschmid/Entitas-CSharp>by Simon Schmid) or recently rewritten Entity Component System in Unity, those aspects are much better thought out and implemented. For example, most often System does not filter entities on its own, but receives dynamically managed, already filtered list of entities (appropriately updated underneath when entities are adding or removing components). The presented API is also far from perfect. In addition, a mature ECS implementation must support communication between the systems and the relationships between them (supported by some kind of messaging system), which is completely omitted here.

Entity Component System is overwhelmingly popular in game development, but I believe it may be justified in high-performance scenarios where data-oriented design makes sense. Having a lot of different “entities” with various characteristics, which need to be processes in huge batches? Does not that sound like ECS?

## **More on Future...**

This section contains a list of features that probably could be included in any other part of this chapter (or previous one) because they are quite general. I decided to gather them in a common “future” section because at the time of writing, they are planned for less or more distant future releases of .NET. Probably at the time of your reading some or most of them are already available or even yet already well-established in the .NET ecosystem. On the other hand, seeing absorption of other newer types (like already available Span<T>), at least a few years will pass before they settle in the widespread awareness of programmers.

### **NULLABLE REFERENCE TYPES**

Nullable reference types may, but are not guaranteed, to be introduced in C# 8.0. Although they are not directly related to memory management - their usage does not incur better or worse performance or memory consumption - they are such important change related to generally understood memory safety, that a book about memory in .NET just cannot simply ignore it.

In the context of null, everyone must cite British computer scientist Tony Hoare who invented a null reference while designing ALGOL language. In 2009 he apologized for inventing it:

*I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn’t resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.*

Is null billion-dollar really a mistake? Could you imagine a world, C# and .NET world, without null and all those NullReferenceException occurrences in your life? Generally, it is hard to imagine a language that does not have any notion of “nothing.” Some values are optional because the domain they come from specifies them as such (being middle name a canonical example). What really null complicates is a lack of clear intent whether it makes sense in a specific context that such “nothing” is allowed (because null is allowed always by default).

Some languages, especially functional ones, replaced nullable types with option types - a polymorphic type that represents an optional value (so it may represent “nothing” or a value). For example, F# uses Option type defined as discriminated union with two cases: Some (containing value) and None. Having such optional type explicitly says that there is possibility a value may be “nothing.” Programmer need to use appropriate checks before accessing such type value (or at least it may be checked by a compiler if she does so).

Ideally, reference types in C# should contain such “optionally nullable” reference types to get rid of current “always nullable” reference types. To have clear intent of nullability, two new kinds of safe reference types are planned to be introduced:

* *nullable reference type* - they may have null assigned so dereferencing them always require checking for null value (and such check may be enforced by C# compiler). Please note they differ from current reference types because while being always nullable, dereferencing them now is not guarded by compiler checks. Such types are representing optional value like Option in F#.
* *non-nullable reference type* - they will never have null value so it is always safe to dereference them.

Of course, care should be taken to introduce them in a way that helps to find bugs in existing code without a need to rewrite everything. To make existing code benefit from them, current reference types must take one of these roles (instead of, for example, introducing two new kinds of reference types besides the existing one). It was decided that current, unannotated reference type will be treated as non-nullable reference type. As Mads Torgersen says on behalf of the whole C# language team, this is because:

* They believe reference types actually requiring null values are less common that we may think.
* C# language already has ? syntax of nullable value types so it seems natural to extend it for reference types.
* It seems right to explicitly express a need of nulls and opt-in for them, rather than the other way around.

So in other words, nullable reference types are going to be added in some future C# version (with the ? syntax) while the behavior of already existing reference types will be been changed into non-nullable reference types (see Listing [14-59](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC59)). This is why this feature is officially called nullable reference types, while we should remember that in fact both new reference types behavior are new.

public class SomeClass

{

public int Field;

public OtherClass? NullableReference; // May be null

public OtherClass NonNullableReference; // May not be null

}

public class OtherClass

{

public int OtherField;

}

***Listing 14-59***An example of a class with both non-nullable (by default) and nullable (by explicitly stating) reference type fields

Obviously, such a change may generate a lot of errors while compiling existing, pre-nullable reference types code. This is by design, however, as those types are introduced to help us with finding null-related bugs in the first place. Not to paralyze the work, it has decided to treat such null-related issues as warnings, instead of errors (while you may still opt-in to errors though).

With this feature, C# compiler does it best to check for nullability violations, especially with respect to local variables and parameters access (see Listing [14-60](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC60)). When accessing nullable object instances without any checks (like in first line in Listing [14-60](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC60)), appropriate warnings are generated. The same happens when compiler discovers null is being accessed (like in the last line in Listing [14-60](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC60)). Program flow control is considered (like conditions and loops) also, as we may see in Listing [14-60](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC60).

public static void UseNullableReference(SomeClass? obj)

{

Console.WriteLine(obj.Field); // Warning CS8602: Possible dereference of a null reference.

Console.WriteLine(obj?.Field); // Ok, checked

if (obj == null)

return;

Console.WriteLine(obj.Field); // Ok, checked above

obj = null;

Console.WriteLine(obj.Field); // Warning CS8602: Possible dereference of a null reference.

}

***Listing 14-60***Compiler behavior with nullable reference type argument

However, there always will be a problem of how deep such a nullability violation check should be. Currently method calls are ignored, as they may contain logic of any complexity you can imagine. So even if ArgumentsValid method checks for null internally (in Listing [14-61](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC61)), a warning still will be generated.

public static void UseChainedNullableReference(SomeClass? obj)

{

if (!ArgumentsValid(obj))

return;

Console.WriteLine(obj.Reference.OtherField); // Warning or not, depending on the check used

}

***Listing 14-61***Compiler behavior with nullable reference type argument

On the other hand, accessing non-nullable reference types is much safer so the compiler will generate many less errors (see Listing [14-62](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC62)).

public static void UseNonNullableReference(SomeClass obj)

{

Console.WriteLine(obj.Field); // Ok

Console.WriteLine(obj?.Field); // Ok, checked

if (obj == null)

return;

Console.WriteLine(obj.Field); // Ok, checked above

obj = null; // Warning CS8600: Converting null literal or possible null value to non-nullable type.

Console.WriteLine(obj.Field); // Warning CS8602: Possible dereference of a null reference.

}

***Listing 14-62***Compiler behavior with non-nullable reference type argument

Warning CS8600 may be surprising though, as it seems we may still assign null to a non-nullable reference type! This is because of many scenarios where it is still necessary (and most of them generate an appropriate warning) - like explicitly assigning null like in Listing [14-62](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC62) or assigning a nullable reference type to non-nullable reference type. There is still one important exception decided to not generate any warnings - an array creation (see Listing [14-63](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC63)). In case of an array of non-nullable types, the compiler should require initialization of all its elements but this would break a lot of existing code. Array declarations like in Listing [14-63](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC63) are overwhelmingly popular so even emitting a warning would flood our compilation with an unmanageable number of messages.

SomeClass[] array = new SomeClass[4];

UseNonNullableReference(array[1]); // Ok, warning is not generated.

***Listing 14-63***Compiler behavior with the array of non-nullable reference type

Please note that at the time of writing this book, nullable reference types are in the pre-release version before official release (planned but not yet confirmed for C# 8.0). This section presents possible design and usage of this feature, to give you an overall picture of why and what it does. Please update your knowledge with official .NET documentation regarding the current state of this feature at the time of reading this book.

What is null by the way? In general, it is a representation of an address that should never happen in normal code, to differentiate it from valid pointers (and references in case of .NET). In all popular programming environments, it is an address of value 0 - because at least the first OS memory page is always kept free (unused) so it is always an invalid address. Being a zero is also useful because pointers and references are becoming null by default in zeroed memory regions (like reference type fields in an object).

Any access to an invalid page (like mentioned on the first page) raises an exception by the OS which is then handled by the CLR. The difference is that if the first page was accessed (which is typically, first 64KB), such exception would be turned into a well-known NullReferenceException. On the other hand, if any higher address was accessed, AccessViolationException will be thrown. So for example, when in C# one tries to access an unmanaged zero pointer, NullReferenceException will occur (see Listing [14-64](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC64)).

unsafe { int read = \*((int\*)IntPtr.Zero); }

***Listing 14-64***Example of unsafe code generating NullReferenceException

On the other hand, if we try to access an address higher than the first 64 KB, AccessViolationException will occur (Listing [14-65](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC65)).

unsafe { int read = \*((int\*)0x1\_0000 + 1); }

***Listing 14-65***Example of unsafe code generating AccessViolationException

Most often NullReferenceException happens in regular C# code, when we try to access a field of null reference (see Listing [14-66](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC66)). This is however handled in the same way because accessing an object’s field is just dereferencing a given address with a small field’s offset (see Listing [14-67](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC67)). In our example, if the reference argument passed in rcx is 0, the corresponding field address will be calculated as 0x8 (assuming Field is the first field in SomeClass). Trying to access 0x8 address still results in NullReferenceException because it fits into the first page.

public static void Test(SomeClass obj)

{

Console.WriteLine(obj.Field);

}

***Listing 14-66***Example of managed code generating NullReferenceException (assuming obj is null)

C.Test(SomeClass)

L0000: sub rsp, 0x28

L0004: mov ecx, [rcx+0x8]

L0007: call System.Console.WriteLine(Int32)

L000c: nop

L000d: add rsp, 0x28

L0011: ret

***Listing 14-67***Assembly code of Test method from Listing [14-66](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC66)

Immediately we may wonder what if an object is bigger than the first page and we are trying to access the end of it (via null reference)? Will it confusingly throw AccessViolationException instead of NullReferenceException ? The answer is, no. Such scenarios are guarded by JIT that generates appropriate code. For example, in case of passing an array, bound-checking code is injected anyway (accessing array’s size field) so it will result in NullReferenceException even before trying to access given element. And if we imagine an enormous object with thousands of fields (see Listing [14-68](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC68)), JIT will add null checking of the entire object before accessing a specific field (see Listing [14-69](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC69)). The second assembly instruction from Listing [14-69](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC69) is generated only when higher fields of SomeClass instance are accessed (if rcx is zero, it will trigger throwing NullReferenceException).

public class SomeClass

{

public long Field0;

public long Field1;

public long Field2;

...

public long Field8229;

public long Field8230;

}

public static void Test(SomeClass obj)

{

Console.WriteLine(obj.Field8000);

}

***Listing 14-68***Example of managed code generating NullReferenceException (assuming obj is null)

C.Test(SomeClass)

L0000: sub rsp, 0x28

L0004: cmp [rcx], ecx

L0006: mov rcx, [rcx+0xfa08]

L000d: call System.Console.WriteLine(Int64)

L0012: nop

L0013: add rsp, 0x28

L0017: ret

***Listing 14-69***Assembly code of Test method from Listing [14-68](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC68)

Please note that both 0 and the first page are used here in terms of virtual memory of a given address. This means that physically “null page” is mapped to some arbitrary physical page.

### **PIPELINES**

Streams are as old as the entire .NET. They are great and do their job but are not well-suited for high-performance code. They may allocate a lot, requiring copying memory here and there. And they introduce overhead of required synchronization when used in multithreading scenarios. For writing efficient code using buffers, like streams, something new has to be invented. This is exactly how *pipelines* (initially called *channels*) were invented, mostly with network streaming kept in mind, used in a new Kestrel web hosting server. But even Kestrel was one of the main reasons behind them, they will be exposed as a general-purpose library.

Upcoming versions of .NET, at the time of this writing, are expected to include completely new API for pipelines, which may be seen as Stream-like buffers that target a range of problems related to high-performance and high-scalable code. They are designed in a producer-consumer manner, so there is a writer (sending data) and a receiver (reading those data). As its current documentation says: “A pipeline is like a Stream that pushes data to you rather than having you pull. One chunk of code feeds data into a pipeline, and another chunk of code awaits data to pull from the pipeline.” As other techniques showed in this chapter, most probably only low-level libraries creators will be interested in them - to be used in networking or serialization code.

Because pipelines are from the ground up designed in high performance and scalability requirements in mind, they have the following characteristics:

* Their memory usage is based on pooling of internal buffers - it allows them to avoid heap allocations.
* They intensively use Span<T> and Memory<T> on API level - it allows them to provide zero-copy usage of the data (data is being provided by slicing internal buffers without a need for copying anything).
* They are asynchronous and thread-safe in an efficient manner.

Regardless of all the complicated machinery underneath, pipeline API is quite straightforward. First of all, we must configure a pipeline instance providing a memory pool that will be used by them (see Listing [14-70](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC70)). There are other configuration options that are not described in this book, especially related to pipe schedulers. This is because my intent is to only briefly describe pipelines capabilities and usage, without going any further with advanced topics. Although they are interesting, this book can’t cover everything in detail.

var pool = MemoryPool<byte>.Shared;

var options = new PipeOptions(pool);

var pipe = new Pipe(options);

***Listing 14-70***Example of pipeline configuration

An instantiated pipeline provides two crucial properties: Writer and Reader. The basic usage of them is presented in Listing [14-71](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC71). Keep in mind that write and read side from such example could be split into two different threads in a thread-safe manner. As we may see, when using pipelines, we must explicitly flush the writer buffers with the help of FlushAsyncs method (to make data visible for readers). And the reader must explicitly update the reading position with the help of AdvanceTo method (to inform pipeline that underlying data has been read so corresponding buffers may be released).

static async Task AsynchronousBasicUsage(Pipe pipe)

{

// Write data

pipe.Writer.Write(new byte[] { 1, 2, 3 }.AsReadOnlySpan());

await pipe.Writer.FlushAsync();

// Read data

var result = await pipe.Reader.ReadAsync();

byte[] data = result.Buffer.ToArray();

pipe.Reader.AdvanceTo(result.Buffer.End);

data.Print();

}

***Listing 14-71***Basic usage of pipelines

However, while pipelines usage presented in Listing [14-71](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC71) is useful for introductory purposes, it is quite an anti-pattern because:

* writer had to heap-allocate byte array before sending data,
* reader had to heap-allocate byte array where read data were copied.

Obviously, it stands in contradiction with the assumptions that were mentioned at the beginning of this section. To make better use of pipelines features, we may get a buffered memory straight from the pipeline itself.

Let’s start from improving the write side of our example (see Listing [14-72](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC72)). As we can see, we may get buffered Span<byte> or Memory<T> from the Writer directly, which does not require any allocations (underneath a slice of required size is returned to use from internal buffers). After accordingly modifying data in the acquired Span<T>, we must explicitly update the writing position with the help of Advance method. It informs the pipeline how many bytes are considered to be written and will be flushed by the following FlushAsync method .

static void SynchronousGetSpanUsage(Pipe pipe)

{

Span<byte> span = pipe.Writer.GetSpan(minimumLength: 2);

span[0] = 1;

span[1] = 2;

pipe.Writer.Advance(2);

pipe.Writer.FlushAsync().GetAwaiter().GetResult();

var readResult = pipe.Reader.ReadAsync().GetAwaiter().GetResult();

byte[] data = readResult.Buffer.ToArray();

pipe.Reader.AdvanceTo(readResult.Buffer.End);

data.Print();

pipe.Reader.Complete();

}

***Listing 14-72***Usage of pipelines with buffered memory. Because of Span<byte> usage, method is not async

We should conceptually treat data returned by GetSpan and GetMemory methods as separate blocks that will be written into the pipeline. Those blocks have a configurable minimum size, which is 2,048 bytes by default. So even if we ask for minimumLength of a few bytes, we will receive 2 kB of memory (this is not a problem as it uses pool internally so no heap allocations are required). Be aware that the returned memory block most probably is reused and may already contain some previously written data. So it is important that Advance method call will truly say how many bytes were indeed modified. Listing [14-73](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC73) shows two successive writes of two acquired buffered blocks but more bytes were “advanced” that really modified. As a result, some parts of read data may have undefined values (0 is our example).

static async Task AsynchronousGetMemoryUsage(Pipe pipe)

{

Memory<byte> memory = pipe.Writer.GetMemory(minimumLength: 2);

memory.Span[0] = 1;

memory.Span[1] = 2;

Console.WriteLine(memory.Length); // Prints 2048

pipe.Writer.Advance(4);

await pipe.Writer.FlushAsync();

Memory<byte> memory2 = pipe.Writer.GetMemory(minimumLength: 2);

memory2.Span[0] = 3;

memory2.Span[1] = 4;

pipe.Writer.Advance(4); // Prints 2048

await pipe.Writer.FlushAsync();

//pipe.Writer.Complete(); close the pipeline from writer side (so reader will not expect more data)

var readResult = await pipe.Reader.ReadAsync();

byte[] data = readResult.Buffer.ToArray();

pipe.Reader.AdvanceTo(readResult.Buffer.End);

data.Print(); // 1,2,0,0,3,4,0,0

//pipe.Reader.Complete(); no more reads possible

}

***Listing 14-73***Usage of pipelines with buffered memory. Thanks to Memory<byte> usage, method may be async.

Improving the read side of pipeline usage to use a zero-copy approach requires a little more, yet still quite intuitive changes. Instead of aggressively reading all readResult.Buffer data and copying it to a newly created array, we may investigate it and access data without copying. Reader.Buffer is of type ReadOnlySequence<byte> that provides the following features:

* such sequence (buffer) represents one or more segments received from the producer,
* its IsSingleSegment property tells us whether sequence represents only single segment,
* its First property is of ReadOnlyMemory<byte> type and returns the first segment,
* it is enumerable, providing ReadOnlyMemory<byte> elements in case of representing multiple segments.

This leads us to a common way of consuming a read buffer (see Listing [14-74](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC74)). Please note that no allocations happen in the presented code - read data is represented by sliced ReadOnlyMemory<byte> and ReadOnlySpan<byte> structs.

Additionally, one more feature of a pipeline is presented in Listing [14-74](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC74) - reader’s AdvanceTo method may update two different read positions separately:

* consumed position - to inform that memory until such position has been already read (consumed) and we do not need it anymore. Such data will not return to us after successive reader’s ReadAsync calls (and may be released by underlying buffering mechanism).
* examined position - to inform that although we read data until such position (we’ve already seen them) but it was not enough for us – so, for example, we have read only a part of incoming message and we must wait for the rest. Data between consumed and examined position will return to us after successive ReadAsync calls altogether with a new data that arrives.

static async Task Process(Pipe pipe)

{

PipeReader reader = pipe.Reader;

var readResult = await pipe.Reader.ReadAsync();

var readBuffer = readResult.Buffer;

SequencePosition consumed;

SequencePosition examined;

try

{

ProcessBuffer(in readBuffer, out consumed, out examined);

}

finally

{

reader.AdvanceTo(consumed, examined);

}

}

private static void ProcessBuffer(in ReadOnlySequence<byte> sequence, out SequencePosition consumed, out SequencePosition examined)

{

consumed = sequence.Start;

examined = sequence.End;

if (sequence.IsSingleSegment)

{

// Consume buffer as single span

var span = sequence.First.Span;

Consume(in span);

}

else

{

// Consume buffer as collections of spans

foreach (var segment in sequence)

{

var span = segment.Span;

Consume(in span);

}

}

// out consumed - to which position we have already consumed the data (and do not need them anymore)

// out examined - to which position we have already analyzed the data (data between consumed and examined will be provided again when new data arrives)

}

private static void Consume(in ReadOnlySpan<byte> span) // No defensive copy as ReadOnlySpan is readonly struct

{

//...

}

***Listing 14-74***Example of zero-copy read side of pipeline

The way of zero-copy reading from pipelines presented in Listing [14-74](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC74) most probably will become a common design pattern. For example, it is already used in HttpParser class in KestrelHttpServer , already presented partially in Listing [14-6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC6) (see Listing [14-75](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC75)). What such parser needs is to interpret incoming network data line by line. So a design pattern presented in a ProcessBuffer method should be modified to read incoming buffer data, seeking a newline character. If a new line end has been found, the consumed position is set accordingly. But if not, data is mark only as examined so it will be reinterpreted once again when new data comes.

public unsafe bool ParseRequestLine(TRequestHandler handler, in ReadOnlySequence<byte> buffer, out SequencePosition consumed, out SequencePosition examined)

{

consumed = buffer.Start;

examined = buffer.End;

// Prepare the first span

var span = buffer.First.Span;

var lineIndex = span.IndexOf(ByteLF);

if (lineIndex >= 0)

{

consumed = buffer.GetPosition(lineIndex + 1, consumed);

span = span.Slice(0, lineIndex + 1);

}

else if (buffer.IsSingleSegment)

{

// No request line end

return false;

}

else if (TryGetNewLine(buffer, out var found))

{

span = buffer.Slice(consumed, found).ToSpan();

consumed = found;

}

else

{

// No request line end

return false;

}

// Fix and parse the span

fixed (byte\* data = &MemoryMarshal.GetReference(span))

{

ParseRequestLine(handler, data, span.Length);

}

examined = consumed;

return true;

}

private static bool TryGetNewLine(in ReadOnlySequence<byte> buffer, out SequencePosition found)

{

var byteLfPosition = buffer.PositionOf(ByteLF);

if (byteLfPosition != null)

{

// Move 1 byte past the \n

found = buffer.GetPosition(1, byteLfPosition.Value);

return true;

}

found = default;

return false;

}

***Listing 14-75***Full code of ParseRequestLine from HttpParser class from KestrelHttpServer

Interpretation of incoming segments from the read buffer is quite tedious. We need to maintain the interpretation state and correctly handle the interpretation of successive segments (as byte data we interpret most probably will be split into multiple segments). For common scenarios of interpreting underlying segments as stream of bytes, BufferReader helper class is also introduced (see Listing [14-76](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC76)). Underneath it handles interpreting successive segments while providing single and contiguous stream of bytes accessible by Read method. Obviously, it still does not heap allocate anything as it is also based on zero-copy approach internally.

private static void ProcessWithBufferReader(in ReadOnlySequence<byte> sequence, out SequencePosition consumed, out SequencePosition examined)

{

var byteReader = BufferReader.Create(sequence);

while (!byteReader.End)

{

var ch = byteReader.Read();

// Consume... read more, and so on, so forth.

// setting:

consumed = byteReader.Position;

examined = byteReader.Position; // or less if Peek was used

// return if you are done with some part

}

}

***Listing 14-76***An example of BufferReader helper class usage

## **Summary**

We have covered quite a lot of various topics in this chapter. It is a kind of all-in-one bag where seemingly unrelated techniques and types were discussed. In my opinion, however, they have one important thing in common - they are advanced, highly specialized things required mostly in even-more specialized code with high-performance requirements. This is exactly why this chapter has a title “Advanced Techniques,” right?

Many words were spoken here about types like Span<T> or Memory<T>, which allow us to write very efficient, no heap-allocating code as was well as other possibilities, like Unsafe class.

Eventually, we took a little insight into the future of C# and .NET. Of course, predicting the future is always hard. So, I refrained from going too far into the future. Two features that are most important from a memory management perspective were briefly described - nullable reference types and pipelines (one should count here also UTF8 strings that are planned to be introduced).

There are no Rules defined in this chapter. If I were to mention a general one, it would sound: do not over-engineer. I mean, most of the techniques described in this chapter are relevant only on low-level code that should most probably belong to something called Infrastructure Level - preferably generalized and sealed in library or NuGet package. Do not clutter Business Layer with strictly technical types like Span<T> or Memory<T>. They do not belong to the business domain for sure and expressiveness of the domain is one of the most important factors during our application’s domain modeling. Span<T> and Memory<T> are the best types for no-copy handling where performance is critical for advanced scenarios.

**Footnotes**

[1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn1_source)

Originally, this type was even supposed to be called Slice, not Span.

[2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn2_source)

Although specific Stream implementation may implement its buffering and flushing mechanisms, this is used for example purposes. In fact, such design is used in classes like FileStream where stream is replaced by native OS calls.

[3](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn3_source)

Unless we have passed an unmanaged address, see ReturnNativeAsSpan method in Listing [14-5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#PC5).

[4](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn4_source)

But taking Domain Driven Design into consideration, it would be probably even more complex, with separate types to represent money or other data.

[5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn5_source)

Avalon is a codename for WPF engine.

[6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml#Fn6_source)

Please note that triviality of presented processing is for brevity, but it does not change the overall presented approach.

© Konrad Kokosa 2018

Konrad Kokosa*Pro .NET Memory Management*

<https://doi.org/10.1007/978-1-4842-4027-4_15>

# 15. Programmatical APIs

Konrad Kokosa[1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Aff2)

(1)

Warsaw, Poland

This is the last chapter of this book. We have seen, so far, many various topics related to .NET memory management - including a comprehensive description of how, in fact, Garbage Collector in .NET works. Other important topics were also described, including resource management with the help of finalization and disposable objects, various types of handles, usage of structs or many diagnostic scenarios, and practical advice related to all of that. At this moment we should feel quite comfortable in the memory management topic, although the amount of knowledge could be a little overwhelming so going back to at least some parts of the book is fully understandable and advisable.

What’s left then? Not so much indeed. In this chapter I would like to describe a few programmatical APIs related to the GC. They are available from code on different levels, providing different levels of flexibility. I believe it is a good theme for the end of the book. Already more or less understanding the operation of the GC, we can now look at how it can be controlled and measured from code. We start from reviewing an already well-known GC class, mainly for reference, as most of the available methods were already used here and there throughout the book. Then, the CLR Hosting feature is described. Eventually, two great libraries that provide deep diagnostic capabilities are shown - ClrMD and EventTrace. As the crème de la crème, a few words are dedicated to the possibility of changing the whole GC into our custom one.

## **GC API**

As said, a static GC class with its static methods has been quite intensively already used in the previous chapters. Here, I want to briefly summarize its usage and show those little possibilities not yet mentioned or described with insufficient details. I do not repeat myself, so if examples of a specific method usage were already presented, I just refer back to them. All methods were organized into some functional groups, presented as subsections. Moreover, besides the GC class itself, a few other methods and types are presented that perfectly suit the overall “Programmatical GC API” section.

### **COLLECTION DATA AND STATISTICS**

The first group contains properties and methods that inform us about the GC status and internal state of memory.

#### ***GC.MaxGeneration***

This informs about the number of maximum generations currently implemented in the GC. It is mostly useful in a code that would like to iterate over all available generations (to not hard-code its number) - like by successive calls of GC.CollectionCount presented below. Or when you want to check with the help of GC.GetGeneration method whether an object is already in the oldest generation (such usage is shown later as well). Please note, this property currently has a value of 2 because the oldest generation 2 and LOH are treated as one (collected together during full GC).

#### ***GC.CollectionCount(Int32)***

This informs about the number of GC occurrences of a specific generation since the program’s beginning. The generation number we ask for should be not less than 0 and not bigger than a value returned by GC.MaxGeneration. Remember that such count is inclusive, so if generation 1 is condemned, both generations 0 and 1 counters are increased. Thus, Listing [15-1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC1) will produce results as shown in Listing [15-2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC2) (each younger generation collection counter includes collections of older generations).

GC.Collect(0);

Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)} {GC.CollectionCount(2)}");

GC.Collect(1);

Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)} {GC.CollectionCount(2)}");

GC.Collect(2);

Console.WriteLine($"{GC.CollectionCount(0)} {GC.CollectionCount(1)} {GC.CollectionCount(2)}");

***Listing 15-1***Illustration of GC.CollectionCount method usage

1 0 0

2 1 0

3 2 1

***Listing 15-2***Results of code from Listing [15-1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC1)

We can use this method for diagnostic and logging from inside our application. However, most popular usage is probably implementing a “smart” explicit GC call only if it does not happen by itself (see Listing [15-3](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC3)). In that way our code that wants to trigger GC will be less aggressive. Recall Chapter [7](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_7_Chapter.xhtml)’s elaboration about explicitly calling GC in general. We could also use such code to periodically check each generation counter to notice that the collection of a given generation has happened recently (thus, allowing us to create a sort of “callback” that is executed after each GC, if checking granularity is small enough).

if (lastGen2CollectionCount == GC.CollectionCount(2))

{

GC.Collect(2);

}

lastGen2CollectionCount = GC.CollectionCount(2);

***Listing 15-3***Conditional explicit GC call if it didn’t happen by itself

#### ***GC.GetGeneration***

This informs about the generation to which the given object belongs. For valid objects on the Managed Heap, it returns value between 0 and GC.MaxGeneration.

It may be used, for example, to create some generation-aware caching policy. Supposing we want to create a pool of objects that are being pinned, it would be good to reuse only objects from the oldest generation, which are most probably living in gen2-only segments. Assuming objects are pinned for a short period of time, pinning in gen2-only segments is less severe because there is much less probability of full GC during that time.

Thanks to the GC.GetGeneration method, we can create such a pool, maintaining a list of already “aged” objects (preferred to be rented from the pool) and another list of younger objects (with the expectation they will become aged at some time). A draft of such pool is presented in Listing [15-4](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC4). If someone wants to rent an object from the pool (by calling Rent method), already aged objects are first checked for availability. If there is none, a list of already maintained younger objects is checked in the RentYoungObject method. If again, there is none currently, a new object is being created via a provided factory method. When an object is being returned to the pool (by calling Return method), its “age” is checked with the help of GC.GetGeneration method and depending on the result, added to the appropriate collection for later reuse. Additionally, Gen2GcCallback class (described in Chapter [12](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_12_Chapter.xhtml)) is used to perform an action on every full GC to maintain both lists - moving those objects that already landed in the oldest generation from the young collection to the aged collection.

public class PinnableObjectPool<T> where T : class

{

private readonly Func<T> factory;

private ConcurrentStack<T> agedObjects = new ConcurrentStack<T>();

private ConcurrentStack<T> notAgedObjects = new ConcurrentStack<T>();

public PinnableObjectPool(Func<T> factory)

{

this.factory = factory;

Gen2GcCallback.Register(Gen2GcCallbackFunc, this);

}

public T Rent()

{

if (!agedObjects.TryPop(out T result))

RentYoungObject(out result);

return result;

}

public void Return(T obj)

{

if (GC.GetGeneration(obj) < GC.MaxGeneration)

notAgedObjects.Push(obj);

else

agedObjects.Push(obj);

}

private void RentYoungObject(out T result)

{

if (!notAgedObjects.TryPop(out result))

{

result = factory();

}

}

private static bool Gen2GcCallbackFunc(object targetObj)

{

((PinnableObjectPool<T>)(targetObj)).AgeObjects();

return true;

}

private void AgeObjects()

{

List<T> notAgedList = new List<T>();

foreach (var candidateObject in notAgedObjects)

{

if (GC.GetGeneration(candidateObject) == GC.MaxGeneration)

{

agedObjects.Push(candidateObject);

}

else

{

notAgedList.Add(candidateObject);

}

}

notAgedObjects.Clear();

foreach (var notAgedObject in notAgedList)

{

notAgedObjects.Push(notAgedObject);

}

}

}

***Listing 15-4***Draft of PinnableObjectPool<T> implementation, preferring to provide objects from the oldest generation

Obviously, PinnableObjectPool<T> presented here is simplified for brevity and does not include such important aspects as cache trimming or multithreading synchronization (especially in AgeObjects method).

There is already mentioned in Chapter [12](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_12_Chapter.xhtml), an internal PinnableBufferCache class in .NET fundamental libraries (CoreFX) that is a real-world implementation of a pool similar to that presented in Listing [15-4](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC4). It includes cache trimming, a lot of care about optimal multithreading access, and another optimization related to managing both objects collections. I strongly recommend that you find a moment to study the code of this class carefully. It is an excellent summary of many of the aspects discussed in this book.

Please note that if we pass an invalid object to GetGeneration method, we should treat its result as undefined (see Listing [15-5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC5)) - for example, current .NET Core implementation will always return 2 in such a case because it assumes that if an object does not belong to an ephemeral segment, it belongs to one of the LOH or gen2 segments.

UnmanagedStruct us = new UnmanagedStruct { Long1 = 1, Long2 = 2 };

int gen = GC.GetGeneration(Unsafe.As<UnmanagedStruct, object>(ref us));

Console.WriteLine(gen);

Output:

2

***Listing 15-5***Passing invalid, stack-allocated object to GC.GetGeneration method

#### ***GC.GetTotalMemory***

This returns the total number of bytes in use, excluding fragmentation, in all generations. In other words, it is a total size of all managed objects on the Managed Heap. This include the size of already unreachable, dead objects if we do not trigger explicit GC before.[1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fn1) As mentioned in Chapter [12](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_12_Chapter.xhtml), where this method implementation was presented (see Listing [12-9](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_12_Chapter.xhtml#PC9)), be aware that when passing true as its forceFullCollection argument, this method may be very costly. In the worst scenario, it may trigger full-blocking GC 20 times trying to get a stable result!

GetTotalMemory method may be used obviously for diagnostic and logging purposes. Its usage in various unit tests and experiments is popular. However, for the purpose of tracking allocations during the test, GC.GetAllocatedBytesForCurrentThread, described later, is a better alternative.

Moreover, be cautious when using this method for memory-based limiting processing, like web request throttling. Because of not counting fragmentation and overall overhead of segments management (for example, committing some segment’s pages in advance), such measure does not reflect precisely the overall pressure of the memory. For such scenarios, it is better to use overall memory measurements provided by the Process class (or at least relate GC.GetTotalMemory result to them). The simple “Hello world” example in Listing [15-6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC6) illustrates the difference (see Listing [15-7](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC7) for results). Objects in the GC Heap are taking around 600 kB of memory. However, private memory usage of the overall process is around 9 MB (while Virtual Memory is obviously bigger, refer to Chapter [2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_2_Chapter.xhtml) for memory categorization in a process).

static void Main(string[] args)

{

Console.WriteLine("Hello world!");

var process = Process.GetCurrentProcess();

Console.WriteLine($"{process.PrivateMemorySize64:N0}");

Console.WriteLine($"{process.WorkingSet64:N0}");

Console.WriteLine($"{process.VirtualMemorySize64:N0}");

Console.WriteLine($"{GC.GetTotalMemory(true):N0}");

Console.Readline();

}

***Listing 15-6***Using GC.GetTotalMemory and various Process memory-related measurements

Hello world!

9,162,752

146,680,064

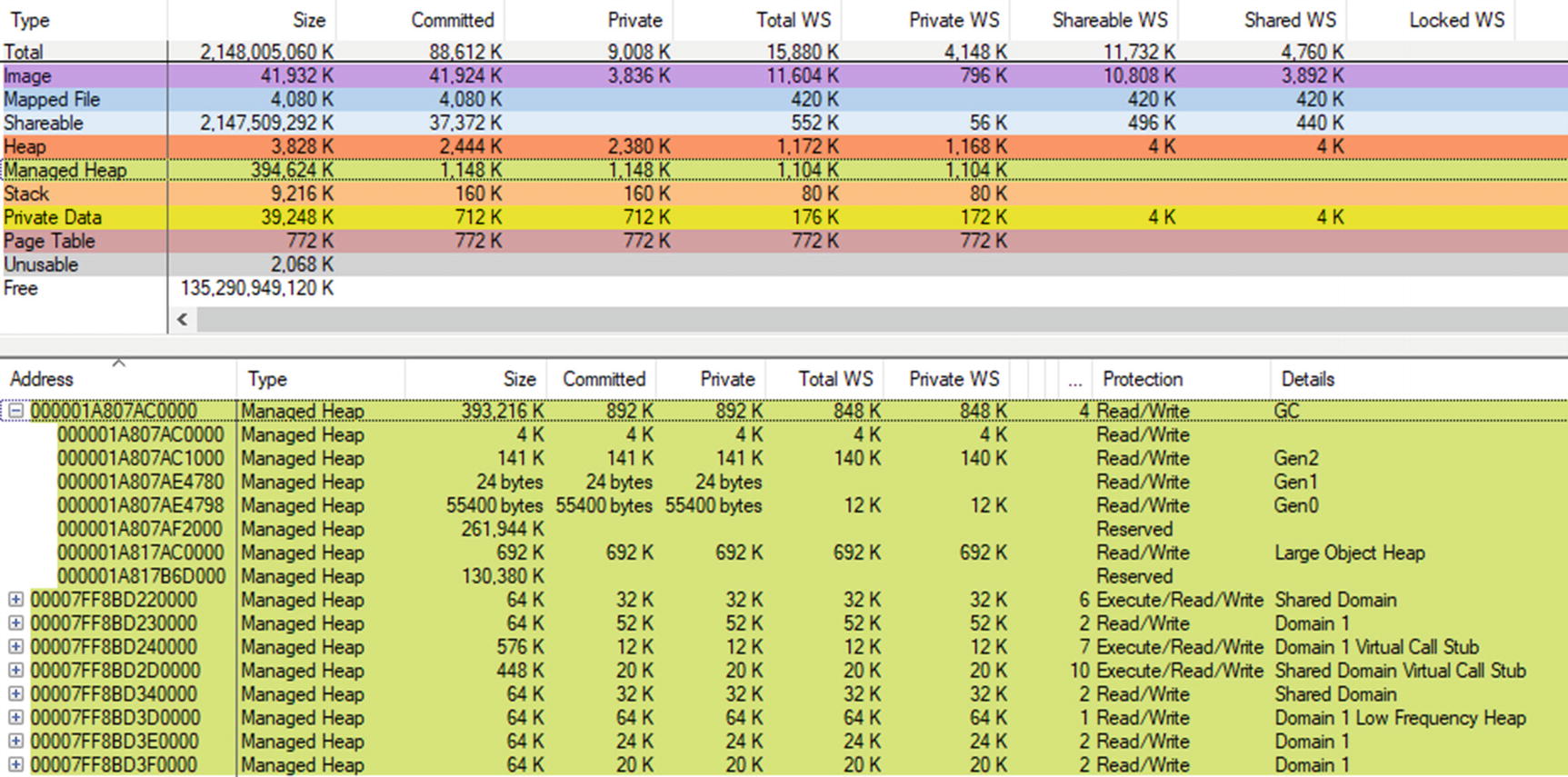
2,199,553,761,280

620,496

***Listing 15-7***Result of code from Listing [15-6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC6)

Even the memory taken by the Managed Heap is noticeably bigger than the total size of objects in it (see Figure [15-1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fig1)). We can see that memory committed by the GC segments take 1,772 kB while results from Listing [15-7](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC7) show only around 600 kB.

And yes, most of this difference lies in fragmentation not being counted in. We may confirm that by using the heapstat command from WinDbg’s SOS extensions (see Listing [15-8](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC8)), where total space taken by free space may be easily calculated.



***Figure 15-1***VMMAP view of program from Listing [15-6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC6) (stopped at the last line)

> !heapstat -inclUnrooted

Heap Gen0 Gen1 Gen2 LOH

Heap0 8216 24 145280 701024

Free space: Percentage

Heap0 24 0 94576 131280 SOH: 61% LOH: 18%

Unrooted objects: Percentage

Heap0 40 0 184 0 SOH: 0% LOH: 0%

***Listing 15-8***HeapStat SOS command result of program from Listing [15-6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC6)

Unfortunately, to get the most interesting Working Set - Private value, you would need to use PerformanceCounter class and read Performance Counters data of your own process. There is also no way to get programmatically overall Managed Heap size including fragmentation other than using ClrMD or ETW-based TraceEvent library presented later in this chapter. There is also an internal GC.GetMemoryInfo method returning such information added in .NET Core 2.1, but at the time of this writing, it was decided to not make it public.

#### ***GC.GetAllocatedBytesForCurrentThread***

This method returns the total number of bytes allocated so far by the current thread. Please note it is a cumulative value and is always growing. It considers only the number of allocations, and it does not matter for this measure how many objects/bytes were afterwards garbage collected.

As it returns a value only for the current thread, it is not possible to ask about allocations on the other thread. Thanks to that, its implementation is fast and straightforward (see Listing [15-9](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC9)): it sums the number of bytes so far allocated in the previous allocation contexts plus the already consumed part of the current allocation context (recall Chapter [5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_5_Chapter.xhtml) where allocation context was described in detail).

FCIMPL0(INT64, GCInterface::GetAllocatedBytesForCurrentThread)

{

...

INT64 currentAllocated = 0;

Thread \*pThread = GetThread();

gc\_alloc\_context\* ac = pThread->GetAllocContext();

currentAllocated = ac->alloc\_bytes + ac->alloc\_bytes\_loh - (ac->alloc\_limit - ac->alloc\_ptr);

return currentAllocated;

}

FCIMPLEND

***Listing 15-9***Implementation of GC.GetAllocatedBytesForCurrentThread method in CoreCLR.

Because the allocation measurement is limited to the only current thread, the GC.GetAllocatedBytesForCurrentThread method is much better suited to isolated unit tests or experiments about allocations, instead of using GC.GetTotalMemory method (see Listing [15-10](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC10)). Please note that the latter provides a total memory usage for overall process so other allocating threads will influence the result. On the other hand, thread isolation in case of this method provides clean and reproducible results.

[Fact]

public void SampleTest()

{

string input = "Hello world!";

var startAllocations = GC.GetAllocatedBytesForCurrentThread();

ReadOnlySpan<char> span = input.AsSpan().Slice(0, 5);

var endAllocations = GC.GetAllocatedBytesForCurrentThread();

Assert.Equal(startAllocations, endAllocations);

Assert.Equal("Hello", span.ToString());

}

***Listing 15-10***Example of using GC.GetAllocatedBytesForCurrentThread in unit test

Please also note this method was added in .NET Core 2.1 and is not available yet in .NET Framework. On the other hand, .NET Framework exposes yet another way of programmatically measuring memory usage with the help of AppDomain class and its two properties[2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fn2):

* MonitoringTotalAllocatedMemorySize - it returns total number of bytes allocated so far by an application domain. It is then similar to the GC.GetAllocatedBytesForCurrentThread method, but it works on the AppDomain, not thread level. Moreover, it is being updated at every allocation context change (which may happen more often than GC). Thus, it has allocation context granularity, which has a few kB accuracies.
* MonitoringSurvivedMemorySize - it returns total number of bytes taken by objects that survived last GC. It is only guaranteed to be accurate after a full GC, although it is updated more often but with less accuracy.

The current mismatch of the methods of allocations measurements causes difficulty when writing code compatible with .NET Standard and designed to be used both by .NET Core and .NET Framework. For example, BenchmarkDotNet library solves this problem using the best possible (most precise) in each case (see Listing [15-11](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC11)).

public struct GcStats

{

private static readonly Func<long> GetAllocatedBytesForCurrentThreadDelegate = GetAllocatedBytesForCurrentThread();

private static Func<long> GetAllocatedBytesForCurrentThread()

{

// for some versions of .NET Core this method is internal,

// for some public and for others public and exposed ;)

var method = typeof(GC).GetTypeInfo().GetMethod("GetAllocatedBytesForCurrentThread",

BindingFlags.Public | BindingFlags.Static)

?? typeof(GC).GetTypeInfo().GetMethod("GetAllocatedBytesForCurrentThread",

BindingFlags.NonPublic | BindingFlags.Static);

return () => (long)method.Invoke(null, null);

}

private static long GetAllocatedBytes()

{

...

// "This instance Int64 property returns the number of bytes that have been allocated by a specific

// AppDomain. The number is accurate as of the last garbage collection." - CLR via C#

// so we enforce GC.Collect here just to make sure we get accurate results

GC.Collect();

#if CLASSIC

return AppDomain.CurrentDomain.MonitoringTotalAllocatedMemorySize;

#elif NETSTANDARD2\_0

...

// https://apisof.net/catalog/System.GC.GetAllocatedBytesForCurrentThread() is not part of the .NET Standard, so we use reflection to call it..

return GetAllocatedBytesForCurrentThreadDelegate.Invoke();

#elif NETCOREAPP2\_1

// but CoreRT does not support the reflection yet, so only because of that we have to target .NET Core 2.1

// to be able to call this method without reflection and get MemoryDiagnoser support for CoreRT ;)

return System.GC.GetAllocatedBytesForCurrentThread();

#endif

}

...

}

***Listing 15-11***Fragments of BenchmarkDotNet’s GcStats class used by MemoryDiagnoser

#### ***GC.KeepAlive***

GC.KeepAlive is a method that extends the liveness of a stack root, because it makes the passed argument reachable at least to the line when this method is called (influencing generated GC info). The use and significance of this method is discussed in Chapter [8](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_8_Chapter.xhtml) (see Listings [8-16](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_8_Chapter.xhtml#PC19) and [8-17](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_8_Chapter.xhtml#PC20)). It was also used in several other examples throughout the book.

#### ***GCSettings.LargeObjectHeapCompactionMode***

By setting this property to GCLargeObjectHeapCompactionMode.CompactOnce value, we may explicitly ask for compacting LOH when the first-blocking full-GC will occur. The usage and performance impact of this settings was thoroughly described in Scenario 10-1- Large Object Heap Fragmentation in Chapter [10](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_10_Chapter.xhtml).

#### ***GCSettings.LatencyMode***

By setting this property, we control the latency mode of the GC, which allows us to control GC’s concurrency and enables additional modes like LowLatency or SustainedLowLatency. The usage of various latency modes and elaboration of which one we should choose was presented in Chapter [11](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_11_Chapter.xhtml).

#### ***GCSettings.IsServerGC***

This indicates whether CLR was started with Workstation or Server GC mode (see Chapter [11](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_11_Chapter.xhtml)). Please note this is a read-only property as the GC mode cannot be changed after runtime has been started. This field value is also not affected by any other settings, like latency mode. Altogether with the pointer size (designating bitness of a process) and the number of processors, it may provide quite comprehensive diagnostic data that you may wish to log during application startup (see Listing [15-12](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC12)).

Console.WriteLine("{0} on {1}-bit with {2} CPUs",

(GCSettings.IsServerGC ? "Server" : "Workstation"),

((IntPtr.Size == 8) ? 64 : 32),

Environment.ProcessorCount);

***Listing 15-12***Example of getting simple diagnostic data

### **GC NOTIFICATIONS**

Part of the GC API are notifications, which allow us to be notified about the possibility of full, blocking GC. Such need comes mainly from pre-.NET 4.5 times where the Server GC had only the non-concurrent, blocking version. Because such GC could take a while, having the possibility to react on it was quite useful. A typical example is to use such notification to tell the load balancer to make this server instance unavailable for the duration of a full-blocking GC. Nowadays GC notifications have lost their importance as most often web applications are running in Background GC mode, with much less noticeable pause times. Moreover, only blocking garbage collections raises such notifications. Thus, if the concurrent configuration is enabled, background garbage collection will not be emitted.

Notifications API consists of the following methods:

* GC.RegisterForFullGCNotification(int maxGenerationThreshold, int largeObjectHeapThreshold) - registers GC notification that should be raised if conditions are met to full-blocking GC make this happen. Those conditions are based on generation 2 or LOH allocation budgets utilization It is then important to remember that those notifications are not directly related to the real GC. As MSDN says: “Note that the notification does not guarantee that a full garbage collection will occur, only that conditions have reached the threshold that are favorable for a full garbage collection to occur.” If we specify too high of values, we will get a lot of false positive notifications that do not come before real GC. On the other hand, if we specify too low of values, we may miss real GCs that happened.
* GC.CancelFullGCNotification - cancels the registration of GC notification.
* GC.WaitForFullGCApproach - it is a blocking call that waits indefinitely for GC notification (there is also method overload with a parameter to specify a timeout value).
* GC.WaitForFullGCComplete - it is a blocking call that waits indefinitely for full-GC being completed (and again, there is method overload with a parameter to specify a timeout value).

A typical example of GC notifications usage is presented in Listing [15-13](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC13). One of the dedicated threads is periodically waiting for GC notification and takes appropriate action if it happens.

GC.RegisterForFullGCNotification(10, 10);

Thread startpolling = new Thread(() =>

{

while (true)

{

GCNotificationStatus s = GC.WaitForFullGCApproach(1000);

if (s == GCNotificationStatus.Succeeded)

{

Console.WriteLine("GC is about to begin");

}

else if (s == GCNotificationStatus.Timeout)

continue;

// ...

// react to full GC, for example call code disabling current server from load balancer

// ...

s = GC.WaitForFullGCComplete(10\_000);

if (s == GCNotificationStatus.Succeeded)

{

Console.WriteLine("GC has ended");

}

else if (s == GCNotificationStatus.Timeout)

Console.WriteLine("GC took alarming amount of time");

}

});

startpolling.Start();

GC.CancelFullGCNotification();

***Listing 15-13***Example of using GC notifications

Remember that this API isn’t exact by design because you are asking to predict the future. Therefore, it requires experimentation with your workload to find appropriate values of GC.RegisterForFullGCNotification arguments.

One could complain about necessity of guessing thresholds provided to RegisterForFullGCNotification, but there are no good alternatives in fact. The situation changes all the time in a real-world process so if it does not happen to be completely regular, it is hard to expect that we will predict future accurately. Fine-tuning with the help of mentioned thresholds allows us at least to adapt to our typical workload.

### **CONTROLLING UNMANAGED MEMORY PRESSURE**

By calling the following methods, we may inform GC that some managed objects are holding (or releasing) some amount of unmanaged memory not directly visible to it:

* GC.AddMemoryPressure(Int64)
* GC.RemoveMemoryPressure(Int64)

If some threshold of such memory is exceeded, GC will be triggered. As mentioned in Chapter [7](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_7_Chapter.xhtml), altogether with those methods’ usage in Scenario 7-3 - Analyzing the Explicit GC Calls, currently this threshold starts at value of 100,000 bytes and is later on dynamically tuned. Listing [12-3](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_12_Chapter.xhtml#PC3) in Chapter [12](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_12_Chapter.xhtml) is yet another typical example of this method usage.

Note also that you could implement your own similar mechanism, if you want, because the default implementation works poorly for you. Although exposed by GC class, this mechanism is not internal to the GC (while still implemented in runtime).

### **EXPLICIT COLLECTION**

The possibility of explicitly calling GC was thoroughly described already in Chapter [7](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_7_Chapter.xhtml). Please refer to the “Explicit Trigger” section in Chapter [7](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_7_Chapter.xhtml) for more details, as well as above-mentioned Scenario 7-3 - Analyzing the Explicit GC Calls.

Just for completeness, please find the list of GC method overloads used to induce such explicit collection:

* Collect()
* Collect(int generation)
* Collect(int generation, GCCollectionMode mode)
* Collect(int generation, GCCollectionMode mode, bool blocking)
* Collect(int generation, GCCollectionMode mode, bool blocking, bool compacting)

### **NO-GC REGIONS**

Regions of code within which runtime tries to disallow GC may be created with the help of the following methods:

* GC.TryStartNoGCRegion(long totalSize)
* GC.TryStartNoGCRegion(long totalSize, bool disallowFullBlockingGC)
* GC.TryStartNoGCRegion(long totalSize, long)
* GC.TryStartNoGCRegion(long totalSize, long lohSize, bool disallowFullBlockingGC)
* GC.EndNoGCRegion()

Further discussion, explanation, and examples of those methods’ usage were already presented in the “No GC Region” section in Chapter [11](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_11_Chapter.xhtml).

### **FINALIZATION MANAGEMENT**

Intimately explained in Chapter [12](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_12_Chapter.xhtml), the set of methods in GC API allow us to control finalization behavior. Such API consists of three methods:

* GC.ReRegisterForFinalize(object obj)
* GC.SuppressFinalize(object obj)
* GC.WaitForPendingFinalizers()

### **MEMORY USAGE**

Handling OutOfMemoryException is cumbersome, especially if it happens in the middle of important processing. To proactively avoid such situations, we may use MemoryFailPoint class that tries to guarantee that there is enough memory available before we start our processing of great importance. Remember that there’s no guarantee that you will not get OutOfMemoryException with this API. It’s just a best effort to avoid it.

Usage of this class is plain and simple (see Listing [15-14](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC14)). MemoryFailPoint constructor will throw InsufficientMemoryException if there is less than the required memory available. Due to internal bookkeeping required for multithreaded usage, MemoryFailPoint is a disposable object so we should remember about calling its Dispose method (or use using clause).

try

{

using (MemoryFailPoint failPoint = new MemoryFailPoint(sizeInMegabytes: 1024))

{

// Do calculations

}

}

catch (InsufficientMemoryException e)

{

Console.WriteLine(e);

throw;

}

***Listing 15-14***Simple example of MemoryFailPoint usage

It is important to note that currently only Windows-based runtimes implement this class functionality. In case of other systems, MemoryFailPoint constructor always succeeds.

In case of current Windows implementation MemoryFailPoint checks for the possibility of allocating a specified amount of managed memory in the following steps:

* Whether there is enough virtual address space in general - this should be always true in case of 64-bit huge address space, as well as it is hard to imagine a need of allocating at once more memory than 32-bit virtual address space.
* It explicitly calls full, blocking, and compacting GC to give it an opportunity to free unused segments and compact managed memory usage as much as possible.
* It checks whether there is enough free virtual memory.
* It checks whether there is a need to grow the OS page file to accommodate required memory size.
* It checks whether there is enough contiguous free virtual memory to create a GC segment, if it is needed.

I strongly encourage you to read MemoryFailPoint class source if you are interested in managing free memory space of a process. Internally it uses Win32 API calls to get currently available memory (in private CheckForAvailableMemory method) and Virtual API’s VirtualQuery call to find a contiguous free virtual address region (in private MemFreeAfterAddress method). It has also a private and internal static method GetMemorySettings(out ulong maxGCSegmentSize, out ulong topOfMemory) implemented in runtime that returns the GC segment size and maximum available virtual address of a process. Relying on such implementation detail, we could even use it to gain information about the segment’s size by the following Reflection usage:

var args = new object[2];

var mi = typeof(MemoryFailPoint).GetMethod("GetMemorySettings", BindingFlags.Static | BindingFlags.NonPublic); mi.Invoke(null, args); // As a result, args[0] contains maxGCSegmentSize value

### **INTERNAL CALLS IN THE GC CLASS**

Just in case you are curious, static GC class is mainly a thin wrapper around intrinsic, runtime method implementations. Most of its methods are marked as InternalCall (see Listing [15-15](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC15)), which are mapped to appropriate runtime methods in CoreCLR’s .\src\vm\ecalllist.h file (see Listing [15-16](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC16)).

public static class GC

{

[MethodImplAttribute(MethodImplOptions.InternalCall)]

public static extern int GetGeneration(Object obj);

[MethodImplAttribute(MethodImplOptions.InternalCall)]

internal static extern bool IsServerGC();

...

}

***Listing 15-15***Fragments of GC class implementation from CoreFX source code

FCFuncStart(gGCInterfaceFuncs)

FCFuncElement("IsServerGC", SystemNative::IsServerGC)

FCFuncElement("GetGeneration", GCInterface::GetGeneration)

...

FCFuncEnd()

***Listing 15-16***Fragments of GC class runtime interface from CoreCLR source code

Static GCInterface methods are calling (mostly) methods defined in gc.cpp file (see Listing [15-17](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC17)).

FCIMPL1(int, GCInterface::GetGeneration, Object\* objUNSAFE)

{

FCALL\_CONTRACT;

if (objUNSAFE == NULL)

FCThrowArgumentNull(W("obj"));

int result = (INT32)GCHeapUtilities::GetGCHeap()->WhichGeneration(objUNSAFE);

FC\_GC\_POLL\_RET();

return result;

}

FCIMPLEND

***Listing 15-17***Example runtime implementation of GC method

## **CLR Hosting**

Whole CLR runtime may be seen as a set of libraries that are able to load and execute CIL code from compatible .NET assembly. Indeed, every time we use .NET, such runtime must be hosted in some process. In case of a regular .NET Framework, thanks to native Windows support, such a host “bootstrap” is contained in the EXE file itself. In case of .NET Core, there is also already a well-known dotnet host application. If we build CoreCLR on our own, there will be also simplified for testing a CoreRun host available. All those hosts have one thing in common - they load the appropriate CLR runtime into process memory, configure it, and execute loaded assembly code (specified from appropriate assembly file). Such host is also included, for example, in SQL Server instance to allow managed code execution from inside it.

Hosting API is publicly exposed and everyone could write its own CLR hosting process. We can imagine many various use cases, but there at least two common ones:

* Create an internal CLR runtime to be able to call managed code from a native process - which is in fact a use case of SQL Server.
* Create customized CLR runtime to gain control over how the CLR works, including the GC.

Because CLR hosting provides many configuration capabilities, we can somehow craft our “own runtime,” suitable for our needs. This is obviously very rarely necessary, so I will not create a full CLR hosting tutorial here. This functionality is pretty well documented. Instead, let’s see a few examples for what it can be used for in the context of memory management.

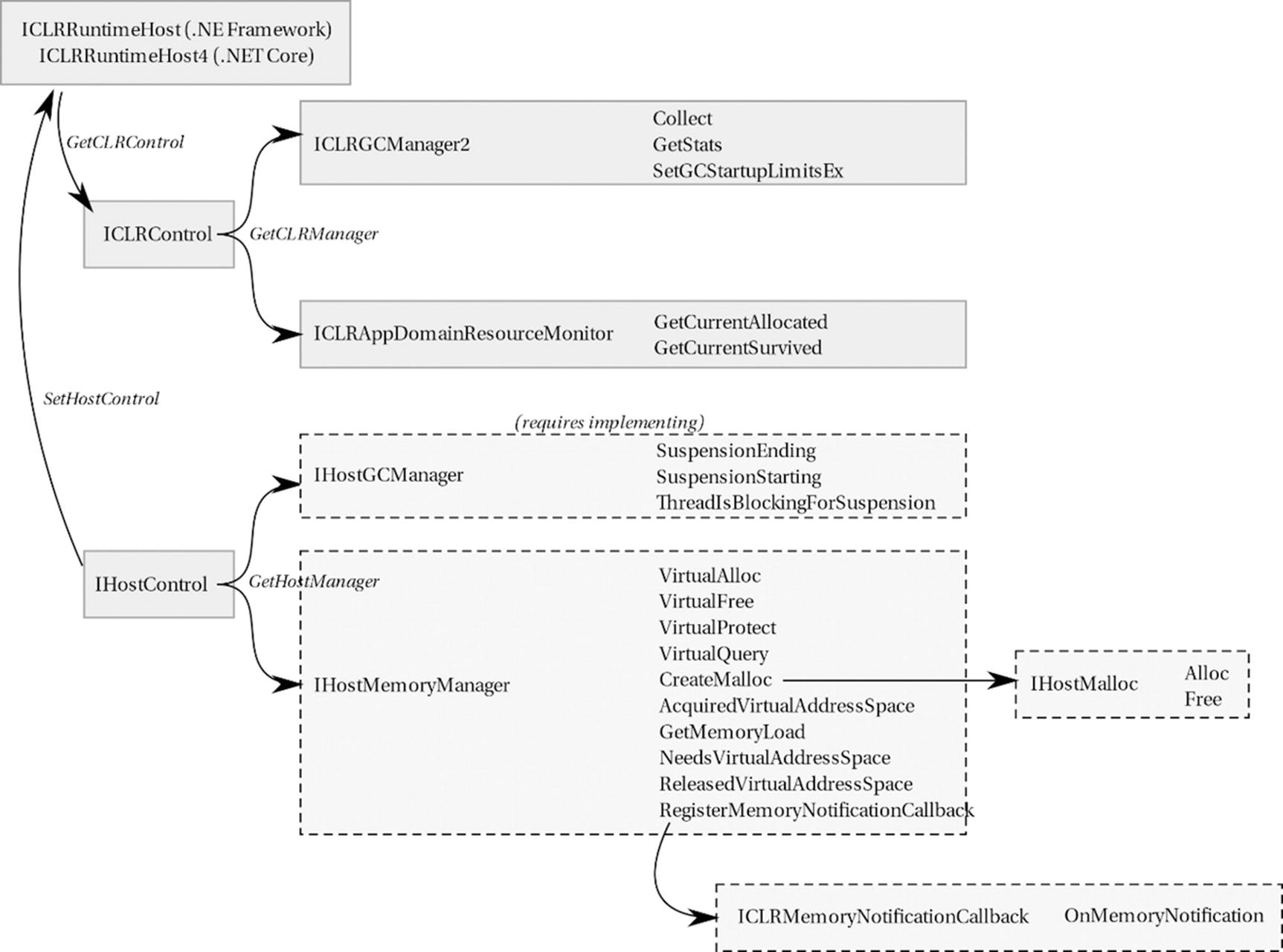
When using CLR Hosting API, we are entering the C++ and COM world - full of well-defined interfaces with well-specified functionality. Every object in CLR Hosting API is represented by some specific interface. The main one, representing the runtime itself, is called ICLRRuntimeHost (in .NET Framework) or ICLRRuntimeHost4 (in .NET Core).[3](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fn3)

CLR Hosting API is slightly different in .NET Framework and .NET Core. Because currently .NET Core version does not support many features interesting to us, only full .NET Framework examples are shown here. Refer to MSDN documentation to see the current status and API of .NET Core version. Currently .NET Core version of CLR Hosting mainly supports loading runtime and executing code, without the possibility of customizing it via the interfaces described below.

Before moving into examples, let’s briefly skim a list of CLR hosting interfaces related to the memory management (including some general, always used ones) to see what is possible in the field of memory management. Although all this information is available on MSDN, I’ve decided to include here a brief summary because it takes a while to merge all this information (including omitting already obsolete interfaces, and so on, and so forth). Currently, from our perspective, the most interesting interfaces are as follows:

* ICLRControl - interface to get various managers, representing specific functionality (like GC, Debugging, Assembly management, and so on, so forth). With respect to .NET memory management, two managers are interesting: ICLRGCManager2 and ICLRAppDomainResourceMonitor.
* ICLRGCManager2 - interface representing some control over GC. More specifically, it includes the following methods:
  + Collect - triggers GC explicitly.
  + GetStats - gets a set of current statistics about the garbage collection - they are directly based on the same values as represented by corresponding performance counters (thus, in CoreCLR build those stats are not available).
  + SetGCStartupLimitsEx - sets the size of GC segment and the maximum size of the generation 0 used during runtime initialization.
* ICLRAppDomainResourceMonitor - it provides measurements about AppDomain, - the same values as MonitoringTotalAllocatedMemorySize and MonitoringSurvivedMemorySize properties of AppDomain object.
* IHostControl - interface allowing to inject various “host managers” into hosted CLR. From a memory management perspective, there are two interesting: IHostGCManager and IHostMemoryManager. If we want to inject our own manager, we have to override GetHostManager method appropriately, returning our custom implementation of those interfaces.
* IHostGCManager - interface providing notifications about GC suspensions, with the following methods that we have to implement:
  + SuspensionStarting - fired when CLR started to suspend threads because of GC.
  + SuspensionEnding - fired when CLR resumed suspended threads because GC of given generation has ended.
  + ThreadIsBlockingForSuspension - fired from each running thread before it is being suspended.
* IHostMemoryManager - interface providing a range of important methods related to memory management. By implementing it, we gain full control over how CLR is consuming system memory for its purposes. We can, for example, change it completely from using Window’s Virtual API to some other libraries (or modify how Virtual API is used). The following methods have to be implemented:
  + AcquiredVirtualAddressSpace - informs that CLR has acquired the specified amount of memory from the operating system. It will not be called if we create our custom memory manager if we omit calling it explicitly.
  + CreateMalloc - allows to get an IHostmalloc interface implementation responsible for requesting heap memory allocations from inside CLR. In this way we can completely change how memory is being allocated for CLR’s internal purposes - for example, replacing default malloc calls with jemalloc memory allocator (mentioned in Chapter [14](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_14_Chapter.xhtml)). Please note this is the internal runtime’s allocator used to allocate memory for private CLR data. It does not replace the GC allocator used to allocate managed objects on the Managed Heap.
  + GetMemoryLoad - returns the amount of physical memory that is currently being used.
  + NeedsVirtualAddressSpace - informs the host that CLR will need specified amount of memory.
  + RegisterMemoryNotificationCallback - allows us to register ICLRMemoryNotificationCallback interface implementation, which is used to notify the CLR on the high memory utilization.
  + ReleasedVirtualAddressSpace - informs the host that CLR will no longer need specified amount of memory.
  + VirtualAlloc - used to acquire virtual memory from the system. Thanks to this method, we may replace or modify how CLR utilizes Virtual API to get memory pages.
  + VirtualFree - used to release virtual memory to the system.
  + VirtualProtect - used to change protection of a given virtual memory region.
  + VirtualQuery - used to query information about given virtual memory region.
* IHostMalloc
  + Alloc - called by the runtime, asking the host to allocate the requested amount of memory from the heap.
  + DebugAlloc - like above but additionally is should track where the memory was allocated.
  + Free - called by the runtime to free memory that was allocated by using the Alloc or DebugAlloc methods .

An overview of how all those relevant interfaces cooperate is presented in Figure [15-2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fig2). Summarizing what is most relevant to us, in our custom CLR host we can override how runtime acquires both memory pages and memory from an unmanaged heap.



***Figure 15-2***The most relevant memory-related interfaces in CLR Hosting API

There are many other possibilities when using custom CLR hosting, but only the most relevant to us were presented. For example, it is possible to take an action on StackOverflowException via ICLROnEventManager. Please also note that .NET Framework before version 2.0 used another set of interfaces, starting from ICorRuntimeHost representing runtime and IGCHost used to control GC. Those interfaces are not described here for brevity as they are rather ancient and no longer used.

An example of loading CLR runtime and obtaining ICLRRuntimeHost and ICLRControl interfaces is presented in Listing [15-18](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC18)[4](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fn4). Remember that presented CLR Hosting examples are written in unmanaged C++ code (and the provided example project is created as a regular Windows console application).

ICLRRuntimeHost\* runtimeHost;

ICLRMetaHost \*pMetaHost = nullptr;

ICLRRuntimeInfo \*pRuntimeInfo = nullptr;

hr = CLRCreateInstance(CLSID\_CLRMetaHost, IID\_ICLRMetaHost, (LPVOID\*)&pMetaHost);

hr = pMetaHost->GetRuntime(L"v4.0.30319", IID\_PPV\_ARGS(&pRuntimeInfo));

hr = pRuntimeInfo->GetInterface(CLSID\_CLRRuntimeHost, IID\_ICLRRuntimeHost, (LPVOID\*)&runtimeHost);

ICLRControl\* clrControl;

hr = runtimeHost->GetCLRControl(&clrControl);

***Listing 15-18***Initialization of CLR Hosting

From now on, we could simply start the runtime and execute the specified method from a given file (see Listing [15-19](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC19)). However, it is the possible customization that interests us the most, so let’s look at some further examples.

DWORD dwReturn;

hr = runtimeHost->Start();

hr = runtimeHost->ExecuteInDefaultAppDomain(targetApp, L"HelloWorld.Program", L"Test", L"", &dwReturn);

***Listing 15-19***Executing code in CLR Hosting

From a CLR memory management point of view, we can distinguish possibilities presented by the CLR hosting into two or three groups:

* configuration - besides providing standard CLR flags (GC workstation/server mode and concurrency), we can tune GC a little by using ICLRGCManager2::SetGCStartupLimitsEx that allows us to set default GC segment size and maximum generation 0 size (see Listing [15-20](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC20)).
* getting diagnostic measurements - thanks to ICLRGCManager2::GetStats or ICLRAppDomainResourceMonitor interface, we may observe memory utilization of hosted CLR instance (see Listing [15-21](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC21)). This may be especially useful in high environments hosting (like production) to observe if hosted managed code does not violate given memory thresholds.
* customization - thanks to IHostControl interface, we may inject a wide range of managers by providing our custom implementations (see Listing [15-22](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC22)). This is the most interesting part of this section so let’s look at this possibility in detail.

ICLRGCManager2\* clrGCManager;

hr = clrControl->GetCLRManager(IID\_ICLRGCManager2, (void\*\*)&clrGCManager);

SIZE\_T segmentSize = 4 \* 1024 \* 1024 \* 1024;

SIZE\_T maxGen0Size = 4 \* 1024 \* 1024 \* 1024;

hr = clrGCManager->SetGCStartupLimitsEx(segmentSize, maxGen0Size);

***Listing 15-20***Example of setting SetGCStartupLimitsEx in CLR Hosting

\_COR\_GC\_STATS gcStats;

gcStats.Flags = COR\_GC\_COUNTS | COR\_GC\_MEMORYUSAGE;

// Based on perf counters so does not work in CoreCLR

hr = clrGCManager->GetStats(&gcStats);

cout << gcStats.CommittedKBytes << endl

<< gcStats.Gen0HeapSizeKBytes << endl

<< gcStats.Gen1HeapSizeKBytes << endl

<< gcStats.Gen2HeapSizeKBytes << endl

<< gcStats.LargeObjectHeapSizeKBytes << endl

<< gcStats.ExplicitGCCount << endl

<< gcStats.GenCollectionsTaken[0] << endl

<< gcStats.GenCollectionsTaken[1] << endl

<< gcStats.GenCollectionsTaken[2] << endl;

***Listing 15-21***Example of getting CLR memory usage data in CLR Hosting

CustomHostControl customHostControl;

hr = runtimeHost->SetHostControl(&customHostControl);

***Listing 15-22***Setting custom host controller in CLR Hosting

Custom IHostControl has to implement GetHostManager method called by CLR for obtaining necessary managers (see Listing [15-23](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC23)). If this method returns E\_NOINTERFACE, the default manager will be used. In our case we want to override IHostMemoryManager implementation to return our CustomHostMemoryManager class. Please note that all COM interfaces should implement also common IUnknown methods: AddRef, Release, and QueryInterface. There are presented here but omitted for brevity in subsequent code listings.

class CustomHostControl : public IHostControl

{

ULONG referenceCounter;

public:

CustomHostControl()

{

referenceCounter = 0;

}

// Inherited via IHostControl

virtual HRESULT GetHostManager(REFIID riid, void \*\* ppObject) override

{

if (riid == IID\_IHostMemoryManager)

{

IHostMemoryManager \*pMemoryManager = new CustomHostMemoryManager();

\*ppObject = pMemoryManager;

return S\_OK;

}

\*ppObject = NULL;

return E\_NOINTERFACE;

}

virtual HRESULT QueryInterface(const IID &riid, void \*\*ppvObject)

{

if (riid == IID\_IUnknown)

{

\*ppvObject = static\_cast<IUnknown\*>(static\_cast<IHostControl\*>(this));

return S\_OK;

}

if (riid == IID\_IHostControl)

{

\*ppvObject = static\_cast<IHostControl\*>(this);

return S\_OK;

}

\*ppvObject = NULL;

return E\_NOINTERFACE;

}

virtual ULONG AddRef()

{

return referenceCounter++;

}

virtual ULONG Release()

{

return referenceCounter--;

}

};

***Listing 15-23***Example of custom IHostControl implementation

Custom HostMemoryManager has the powerful capability of replacing all virtual memory management and heap-allocation handling. Remember that the whole GC (and its internal allocators) is seen as a black box - memory pages will be obtained for it as for any other necessary regions. There is, in fact, no way to distinguish VirtualAlloc call acquiring pages for the Managed Heap from the other calls.

However, even on such a level of customization, we may implement interesting things. For example, we can override VirtualAlloc method to *lock* all acquired pages in physical memory, so they will not be ever paged to disk (with high probability). In such cases, other methods we may leave as thin wrappers around regular a Virtual API (see Listing [15-24](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC24)). Aggressive page locking may improve such .NET application performance as its memory most probably will always reside in the physical RAM.

class CustomHostMemoryManager : public IHostMemoryManager

{

ULONG referenceCounter;

public:

CustomHostMemoryManager() : referenceCounter(0) { }

// Inherited via IHostMemoryManager

virtual HRESULT CreateMalloc(DWORD dwMallocType, IHostMalloc \*\* ppMalloc) override

{

\*ppMalloc = new CustomHostMalloc();

return S\_OK;

}

virtual HRESULT VirtualAlloc(void \* pAddress, SIZE\_T dwSize, DWORD flAllocationType, DWORD flProtect, EMemoryCriticalLevel eCriticalLevel, void \*\* ppMem) override

{

void\* result = ::VirtualAlloc(pAddress, dwSize, flAllocationType, flProtect);

\*ppMem = result;

BOOL locked = false;

if (flAllocationType & MEM\_COMMIT)

{

locked = ::VirtualLock(\*ppMem, dwSize);

}

cout << "VirtualAlloc " << \*ppMem << " (" << dwSize << "), flags: " << flAllocationType << " " << flProtect << " => " << pAddress << " " << locked << endl;

return S\_OK;

}

virtual HRESULT VirtualFree(LPVOID lpAddress, SIZE\_T dwSize, DWORD dwFreeType) override

{

::VirtualFree(lpAddress, dwSize, dwFreeType);

return S\_OK;

}

virtual HRESULT VirtualQuery(void \* lpAddress, void \* lpBuffer, SIZE\_T dwLength, SIZE\_T \* pResult) override

{

\*pResult = ::VirtualQuery(lpAddress, (PMEMORY\_BASIC\_INFORMATION)lpBuffer, dwLength);

return S\_OK;

}

virtual HRESULT VirtualProtect(void \* lpAddress, SIZE\_T dwSize, DWORD flNewProtect, DWORD \* pflOldProtect) override

{

::VirtualProtect(lpAddress, dwSize, flNewProtect, pflOldProtect);

return S\_OK;

}

virtual HRESULT GetMemoryLoad(DWORD \* pMemoryLoad, SIZE\_T \* pAvailableBytes) override

{

// Simulate no problems

\*pMemoryLoad = 1;

\*pAvailableBytes = 1024 \* 1024 \* 1024;

return S\_OK;

}

virtual HRESULT RegisterMemoryNotificationCallback(ICLRMemoryNotificationCallback \* pCallback) override

{

return S\_OK;

}

virtual HRESULT NeedsVirtualAddressSpace(LPVOID startAddress, SIZE\_T size) override

{

return S\_OK;

}

virtual HRESULT AcquiredVirtualAddressSpace(LPVOID startAddress, SIZE\_T size) override

{

return S\_OK;

}

virtual HRESULT ReleasedVirtualAddressSpace(LPVOID startAddress) override

{

return S\_OK;

}

// Inherited via IUnknown

// ...

};

***Listing 15-24***Example of custom host memory manager implementing aggressive page locking in physical memory

Presented custom IHostMemoryManager overrides also CreateMalloc method , which returns our custom IHostMalloc implementation (see Listing [15-25](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC25)). It is shown for illustrative purposes, but we can imagine here a whole set of different implementations, including using the already-mentioned jemalloc library instead of malloc and free functions.

class CustomHostMalloc : public IHostMalloc

{

ULONG referenceCounter;

public:

CustomHostMalloc() : referenceCounter(0) { }

// Inherited via IHostMalloc

virtual HRESULT Alloc(SIZE\_T cbSize, EMemoryCriticalLevel eCriticalLevel, void \*\* ppMem) override

{

\*ppMem = ::malloc(cbSize);

cout << " Alloc " << \*ppMem << " (" << cbSize << ")" << endl;

return S\_OK;

}

virtual HRESULT DebugAlloc(SIZE\_T cbSize, EMemoryCriticalLevel eCriticalLevel, char \* pszFileName, int iLineNo, void \*\* ppMem) override

{

\*ppMem = ::malloc(cbSize);

return S\_OK;

}

virtual HRESULT Free(void \* pMem) override

{

::free(pMem);

return S\_OK;

}

// Inherited via IUnknown

// ...

};

***Listing 15-25***Example of custom heap-allocation implementation for hosted CLR

Such a “non-paged CLR host” as presented here is obviously only a simple draft. Full, much more well-thought-out implementation is already prepared by Sasha Goldshtein and Alon Fliess, currently available at [https://​archive.​codeplex.​com/​?​p=​nonpagedclrhost](https://archive.codeplex.com/?p=nonpagedclrhost). I strongly recommend reading its source code. For example, it takes into consideration limits of possible page locking. Obviously, too aggressive locking could negatively influence overall system performance as other applications will have less physical memory available. As MSDN says: “The maximum number of pages that a process can lock is equal to the number of pages in its minimum working set minus a small overhead.” Thus, Sasha and Alon’s implementation uses SetProcessWorkingSetSize Win32 call to appropriately configure working set limits.

## **ClrMD**

The Microsoft.Diagnostics.Runtime library, also known as ClrMD (or CLR MD) is a set of managed APIs for introspecting managed processes and memory dumps. It is rather designed to build diagnostic tools and small snippets, than to use it as self-monitoring solution of a process (although such possibility also exists as we will soon see). It provides similar capabilities as WinDBG’s SOS extensions but in a much more convenient way available from C# code. Microsoft.Diagnostics.Runtime library is available as a NuGet package and may be used both in .NET Framework and .NET Core applications to analyze both .NET Framework and .NET Core targets. Moreover, full source code of ClrMD is publicly available in GitHub so you can investigate how it is implemented!

Please note that describing all possibilities of this library is not possible here due to book space limitations. The following examples are presented to give you an overall grasp of what is possible and how powerful this library is. Do not treat this section neither as a ClrMD tutorial nor as a comprehensive use-case description. Refer to ClrMD’s documentation and samples for further knowledge.

The root object required to work with ClrMD is DataTarget class instance, which may be obtained by attaching to a running process or loading memory dump, with the help of the following static methods:

* AttachToProcess - allows us to attach to existing process of given PID (Process ID). It may be done in three different ways:
  + Invasive - the process will be paused and we will be able to control it like we attached from the regular debugger. This is a preferred way in normal circumstances.
  + NonInvasive - the process will be paused but we will not be able to control the process. Because in general only a single debugger may control any process, this method is useful if we want to attach to a process with other debugger already attached.
  + Passive - the process in not paused and no debugger is attached to it in any mode. We should be aware that many queries about dynamic data, like thread stacks or object references, may be often inconsistent. The overall idea with this mode is that the program using ClrMD is responsible for doing all process control-related work (like suspending the observed process). This gives the developer complete flexibility in how the target process is controlled.
* LoadCrashDump - allows us to load a file of already taken memory dump (e.g., with the help of ProcDump).

Please note that Passive mode theoretically allows us to attach even to our own process, to provide self-monitoring capabilities. This, however, makes many problems if you think about it deeply - like how ClrMD would handle a dynamically changing state of the process, inspecting a heap while GCs and allocations are happening, and so on, and so forth. Thus, the ClrMD maintainer didn’t specifically disallow self-inspection, because it was something that could be useful in small corner cases. However, doing this correctly is essentially rocket science, not for the faint of heart, and if you run into issues, treat such a scenario as not supported by the maintainer.

When DataTarget is initialized, we may start investigating underlying data, looking for the runtimes that are (or were) used in it (see Listing [15-26](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC26)). This includes information about needed underlying DAC (Data Access Component), which is responsible for understanding all of CLR’s internal data structures.

using (DataTarget target = DataTarget.AttachToProcess(pid, 5000, AttachFlag.Invasive))

{

foreach (ClrInfo clrInfo in target.ClrVersions)

{

Console.WriteLine("Found CLR Version:" + clrInfo.Version.ToString());

// This is the data needed to request the dac from the symbol server:

ModuleInfo dacInfo = clrInfo.DacInfo;

Console.WriteLine($"Filesize: {dacInfo.FileSize:X}");

Console.WriteLine($"Timestamp: {dacInfo.TimeStamp:X}");

Console.WriteLine($"Dac File: {dacInfo.FileName}");

ClrRuntime runtime = clrInfo.CreateRuntime();

...

}

}

***Listing 15-26***Example of simple ClrMD usage - attaching to already running process

Having properly the initialized ClrRuntime instance , we may do a lot of very interesting things. Let’s look at only just a few examples. Please note that only a small part of possible methods or attributes of used ClrMD objects is presented here. Refer to documentation to see all of them.

We may inspect all running threads and print their current stacks (see Listing [15-27](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC27)).

foreach (ClrThread thread in runtime.Threads)

{

if (!thread.IsAlive)

continue;

Console.WriteLine("Thread {0:X}:", thread.OSThreadId);

foreach (ClrStackFrame frame in thread.StackTrace)

Console.WriteLine("{0,12:X} {1,12:X} {2}", frame.StackPointer, frame.InstructionPointer,

frame.ToString());

Console.WriteLine();

}

***Listing 15-27***Example of ClrMD usage - listing all thread’s call stacks

We may iterate through all AppDomains and modules loaded by the runtime, as well as every managed type already used by them (see Listing [15-28](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC28)).

foreach (var domain in runtime.AppDomains)

{

Console.WriteLine($"AppDomain {domain.Name} ({domain.Address:X})");

foreach (var module in domain.Modules)

{

Console.WriteLine($" Module {module.Name} ({(module.IsFile ? module.FileName : "")})");

foreach (var type in module.EnumerateTypes())

{

Console.WriteLine($"{type.Name} Fields: {type.Fields.Count}");

}

}

}

***Listing 15-28***Example of ClrMD usage - listing all AppDomains, modules and types loaded

Please note that ClrMD gives a view into how the runtime sees the process state of the world, and not how things are defined in code. For example, let’s say there’s a module loaded that defines a type Foo, and Foo is never used by the process. In that case, EnumerateTypes may or may not return Foo… depending on whether the runtime decided to load that type out of the module or not. Having said that, whether it does load Foo is an implementation detail that may change from version to version, in the first place.)

However, from our perspective, the most interesting are obviously all memory-related information. For example, we can investigate all memory regions used by CLR, including the Managed Heap (see Listing [15-29](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC29) and sample result in Listing [15-30](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC30)).

foreach (var region in runtime.EnumerateMemoryRegions().OrderBy(r => r.Address))

{

Console.WriteLine($"0x{region.Address:X} (bytes: {region.Size:N0}) - {region.Type} " +

$"{(region.Type == ClrMemoryRegionType.GCSegment ? "(" + region.GCSegmentType.ToString() + ")" : "")}");

}

***Listing 15-29***Example of ClrMD usage - listing all memory regions of a process

0x24198CC1000 (bytes: 4,096) - HandleTableChunk

0x24199541000 (bytes: 200,704) - GCSegment (Ephemeral)

0x24199572000 (bytes: 268,230,656) - ReservedGCSegment

0x241A9541000 (bytes: 69,632) - GCSegment (LargeObject)

0x241A9552000 (bytes: 134,144,000) - ReservedGCSegment

0x7FF9F5250000 (bytes: 12,288) - LowFrequencyLoaderHeap

0x7FF9F5250000 (bytes: 12,288) - LowFrequencyLoaderHeap

0x7FF9F5256000 (bytes: 28,672) - HighFrequencyLoaderHeap

0x7FF9F5256000 (bytes: 28,672) - HighFrequencyLoaderHeap

0x7FF9F525D000 (bytes: 12,288) - StubHeap

0x7FF9F525D000 (bytes: 12,288) - StubHeap

0x7FF9F5260000 (bytes: 12,288) - LowFrequencyLoaderHeap

0x7FF9F5263000 (bytes: 40,960) - HighFrequencyLoaderHeap

0x7FF9F5274000 (bytes: 28,672) - CacheEntryHeap

0x7FF9F527D000 (bytes: 192,512) - DispatchHeap

0x7FF9F52AC000 (bytes: 344,064) - ResolveHeap

0x7FF9F5300000 (bytes: 24,576) - IndcellHeap

0x7FF9F5300000 (bytes: 24,576) - IndcellHeap

0x7FF9F5306000 (bytes: 24,576) - CacheEntryHeap

0x7FF9F5306000 (bytes: 24,576) - CacheEntryHeap

0x7FF9F530C000 (bytes: 16,384) - LookupHeap

0x7FF9F530C000 (bytes: 16,384) - LookupHeap

0x7FF9F5310000 (bytes: 155,648) - DispatchHeap

0x7FF9F5310000 (bytes: 155,648) - DispatchHeap

0x7FF9F5336000 (bytes: 237,568) - ResolveHeap

0x7FF9F5336000 (bytes: 237,568) - ResolveHeap

0x7FF9F53B0000 (bytes: 65,536) - LowFrequencyLoaderHeap

***Listing 15-30***Example results of code from Listing [15-28](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC28)

The Managed Heap may be further investigated through ClrHeap class available as ClrRuntime’s Heap property. It allows for iterating over all currently existing managed objects, as well as traversing those object fields and references (see Listings [15-31](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC31) and [15-32](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC32) for the corresponding result).

ClrHeap heap = runtime.Heap;

foreach (var clrObject in heap.EnumerateObjects())

{

if (clrObject.Type.Name.EndsWith("SampleClass"))

ShowObject(heap, clrObject, string.Empty);

}

private static void ShowObject(ClrHeap heap, ClrObject clrObject, string indent)

{

Console.WriteLine($"{indent}{clrObject.Type.Name} ({clrObject.HexAddress}) - gen{heap.GetGeneration(clrObject.Address)}");

foreach (var reference in clrObject.EnumerateObjectReferences())

{

ShowObject(heap, reference, " ");

}

}

***Listing 15-31***Example of ClrMD usage - listing references of some managed type instances

CoreCLR.HelloWorld.SampleClass (24199564fa0) - gen0

CoreCLR.HelloWorld.AnotherClass (24199564fc0) - gen0

CoreCLR.HelloWorld.AnotherClass (24199564fd8) - gen0

CoreCLR.HelloWorld.SomeOtherClass (24199564ff0) - gen0

***Listing 15-32***Example results of code from Listing [15-31](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC31)

Individual GC segments may be also investigated, thanks to ClrHeap’s Segments property. Each such ClrSegment provides various interesting data, including its internal structure, like generations it contains (see Listing [15-33](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC33) and sample result in Listing [15-34](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC34)).

foreach (var segment in heap.Segments)

{

Console.WriteLine($"{segment.Start:X16} - {segment.End:X16} ({segment.CommittedEnd:X16}) Heap#: {segment.ProcessorAffinity}");

if (segment.IsEphemeral)

{

Console.WriteLine($" Gen0: {segment.Gen0Start:X16} ({segment.Gen0Length})");

Console.WriteLine($" Gen1: {segment.Gen1Start:X16} ({segment.Gen1Length})");

if (segment.Gen2Start >= segment.Start &&

segment.Gen2Start < segment.CommittedEnd)

{

Console.WriteLine($" Gen2: {segment.Gen2Start:X16} ({segment.Gen2Length})");

}

}

else if (segment.IsLarge)

{

Console.WriteLine($" LOH: {segment.Start} ({segment.Length})");

}

else

{

Console.WriteLine($" Gen2: {segment.Gen2Start:X16} ({segment.Gen2Length})");

}

foreach (var address in segment.EnumerateObjectAddresses())

{

var type = heap.GetObjectType(address);

if (type == heap.Free)

{

Console.WriteLine($"{type.GetSize(address)}");

}

}

}

***Listing 15-33***Example of ClrMD usage - listing all GC segments of a process

000002551B871000 - 000002551B896730 (000002551B8A2000) Heap#: 0

Gen0: 000002551B871030 (153344)

Gen1: 000002551B871018 (24)

Gen2: 000002551B871000 (24)

***Listing 15-34***Example results of code from Listing [15-32](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC32)

We already know that the GC implementation detail is that segments (representing heaps) are linked to a CPU that handles allocation, marking, and so on. Conceptually, however, ProcessorAffinity field is better thought of as which Heap# it lives in. Essentially, it should have been probably named something like HeapNumber instead of current ProcessorAffinity.

Filling this section with more and more examples seems to be rather redundant. I believe you’ve already noticed the real power of ClrMD. I will just only mention here a few other interesting possibilities:

* enumerating over all objects in fReachable queue with the help of runtime.EnumerateFinalizerQueueObjectAddresses() method,
* enumerating over all handles with the help of runtime.EnumerateHandles(),
* enumerating all current GC roots with the help of heap.EnumerateRoots(),
* enumerating all current stack roots of a given thread,
* getting an address of JITted method’s code (so we may use some disassembler to see its native code).

Quite popular approach to use ClrMD, especially for memory dump analysis, is to use ClrMD from within LINQPad ( [https://www.linqpad.net](https://www.linqpad.net/) ) application. It provides nice scripting capabilities so we can easily utilize ClrMD without a need of using Visual Studio and creating dedicated projects.

Even though it is so powerful, sometimes we may notice that still ClrMD does not publicly expose some desired properties. One of the examples is investigating the current thread’s allocation context. Although such information is known to ClrMD, relevant properties are not directly accessible. We can use Reflection to get them (but remember that there is no guarantee that used properties will not be changed in future versions).

foreach (ClrThread thread in runtime.Threads)

{

var mi = runtime.GetType().GetMethod("GetThread", BindingFlags.Instance | BindingFlags.NonPublic);

var threadData = mi.Invoke(runtime, new object[] {thread.Address});

var pi = threadData.GetType().GetProperty("AllocPtr", BindingFlags.Instance | BindingFlags.Public);

ulong allocPtr = (ulong) pi.GetValue(threadData);

pi = threadData.GetType().GetProperty("AllocLimit", BindingFlags.Instance | BindingFlags.Public);

ulong allocLimit = (ulong) pi.GetValue(threadData);

}

This is an example that digging into ClrMD source code may be beneficial!

If you are like me, you can see with your eyes all these great diagnostic tools that you can write, thanks to such possibilities. And indeed, there are currently many smaller or bigger initiatives (mostly open sourced) to create such tools, created for various reasons. It is not possible to list them all here, but the two most important should be named: Netext and SOSEX. Those WinDbg extensions are written as wrappers around ClrMD. And yes, it is a little ironic that one of the best WinDbg extensions for .NET diagnostics is written in .NET.

If you want to get a current list of tools based on ClrMD (or integrating with it in some way), please look for Tools built on top of CLRMD online list maintained by Matt Warren available at [http://​mattwarren.​org/​2018/​06/​15/​Tools-for-Exploring-.​NET-Internals](http://mattwarren.org/2018/06/15/Tools-for-Exploring-.NET-Internals).

## **TraceEvent Library**

Microsoft.Diagnostics.Tracing.TraceEvent is a .NET library providing collecting and processing capabilities of ETW data. It is a relevant part of the main PerfView’s machinery, exposed now as a separate Nuget package (but its source code is available also as a part of the PerfView repository).

I would rather like to avoid repeating here basic examples of using TraceEvent to not artificially lengthen the book. You can find comprehensive documentation and examples under the address [https://​github.​com/​Microsoft/​perfview/​blob/​master/​documentation/​TraceEvent/​TraceEventProgra​mmersGuide.​md](https://github.com/Microsoft/perfview/blob/master/documentation/TraceEvent/TraceEventProgrammersGuide.md). Let’s just briefly summarize it that TraceEvent library allows us to record ETW session to a file (regular ETL file known from PerfView) and analyze such file afterwards, or just to create and consume ETW session in real time. Every ETW provider may be enabled and its events appropriately consumed.

For the convenience of using most common ETW providers, TraceEvent library provides two strongly-typed parsers already built in into it: ClrTraceEventParser and KernelTraceEventParser (represented by Clr and Kernel properties of Source property of the session). As the former knows how to parse all the Common Language Runtime events, it is very useful also in all GC-related scenarios. We are just consuming then strongly-typed callbacks representing the reaction on events of our interest. Listing [15-35](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC35) shows an example of creating an ETW session that in real time reacts on the GC start and stop events, printing also the GC statistics.

using (var session = new TraceEventSession("SampleETWSession"))

{

Console.CancelKeyPress += (object sender, ConsoleCancelEventArgs cancelArgs) =>

{

session.Dispose();

cancelArgs.Cancel = true;

};

session.EnableProvider(ClrTraceEventParser.ProviderGuid, TraceEventLevel.Verbose, (ulong)ClrTraceEventParser.Keywords.Default);

session.Source.Clr.GCStart += ClrOnGcStart;

session.Source.Clr.GCStop += ClrOnGcStop;

session.Source.Clr.GCHeapStats += ClrOnGcHeapStats;

session.Source.Process();

}

private static void ClrOnGcStart(GCStartTraceData data)

{

Console.WriteLine($"[{data.ProcessName}] GC gen{data.Depth} because {data.Reason} started {data.Type}.");

}

private static void ClrOnGcStop(GCEndTraceData data)

{

Console.WriteLine($"[{data.ProcessName}] GC ended.");

}

private static void ClrOnGcHeapStats(GCHeapStatsTraceData data)

{

Console.WriteLine($"[{data.ProcessName}] Heapstats - {data.GenerationSize0:N0}|{data.GenerationSize1:N0}|{data.GenerationSize2:N0}|{data.GenerationSize3}");

}

***Listing 15-35***Example of TraceEvent usage - using built-in CLR provider parser

Using CLR and kernel parsers with appropriate callbacks makes consuming ETW data trivial and very pleasant. Obviously, we can observe events related to our own process by filtering incoming events by the ProcessID field. It allows us to provide quite deep self-monitoring insight into a process with very low overhead (assuming we will carefully choose how many providers and keywords we enabled to not flood us with the incoming events).

Additionally, with the help of TraceEvent, we can use the ETW ability to record the event’s stack trace. To make it possible, a “higher-level” type of session interpreter must be used, named TraceLog. If interesting events have stacks registration enabled, we may use CallStack() method on received trace data to obtain a collection of stack frames. Please refer to TraceEvent library code samples to see a working example. Remember also that enabling stack trace capturing significantly increases the session overhead so it should be used carefully.

At this point, we have already described all the possibilities how we can monitor the use of the memory of our application from within a process:

* we can observe allocations of each thread by calling GC.GetAllocatedBytesForCurrentThread method (see Listing [15-10](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC10) earlier in this chapter). Obviously, we may build some process-wide statistics built on top of that functionality, gathering data from each thread. Please remember this is only information about allocations and does not inform in any way how much of allocated memory survives. Thus, it does not say anything about overall memory usage of a process. In case of .NET Framework, we can also use AppDomain’s MonitoringTotalAllocatedMemorySize property for the same purpose (see Listing [15-11](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC11) shown earlier).
* We can observe the total size occupied by managed objects (excluding fragmentation) in all generations by calling GC.GetTotalMemory method (see Listing [15-6](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC6)). As already explained, this is a very informative measurement but without consideration of fragmentation and overall memory taken by the Managed Heap, it does not relate greatly to the process memory consumption as seen from the operating system point of view. It is, however, a great way of noticing memory leak, when there are more and more reachable objects on the Managed Heap. We can additionally observe overall process memory usage by Process properties like WorkingSet64 or PrivateMemorySize64, to support GC.GetTotalMemory measurement.
* We can observe .NET CLR Memory Performance Counters of our own process. This provides great insights into a process (generation sizes, virtual memory consumption, and so on, and so forth) provided with at most one-second granularity, which is enough for many use cases. The main drawback is the fact that Performance Counters are supported only on Windows .NET Framework.
* We can observe the GC ETW events with the TraceEvent library. It provides even more precise and deeper insights into a process, because as we have seen many times in this book, ETW provides tremendous amounts of information. The amount of overhead ETW introduces is proportional to the number of events captured. Observing the not so common GC start/end/GCHeapStats events is a reasonable approach to get high-level memory info.
* We can self-attach the ClrMD library to our own process in a passive way, giving ourselves powerful insights into the Managed Heap (including memory organization into segments, objects, and their references, roots, finalization queues, and so on, and so forth). This is a nice diagnostic approach possibility in Debug build, but I would recommend careful consideration before including it in Release builds on production. Remember that self-attaching in passive mode is not supported by the ClrMD maintainers so it is risky and may lead you to strange problems.

## **Custom GC**

Starting from .NET Core 2.1, coupling between Garbage Collector and the Execution Engine itself have been loosened a lot. Prior to this version, the Garbage Collector code was pretty much tangled with the rest of the CoreCLR code. However, .NET Core 2.1 introduces a concept of *Local GC*, which means the runtime can use a GC in its own dll, which means GC is now pluggable. We can plug in our custom GC by setting a single environment variable (see Listing [15-36](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC36)).

set COMPlus\_GCName=f:\GithubProjects\CoreCLR.ZeroGC\x64\Release\ZeroGC.dll

***Listing 15-36***Setting proper environment variable to replace GC implementation

.NET Core, when initializing, notices such an environment variable and will try to load GC code from the specified library instead of default, built-in GC. The custom GC can contain a completely different implementation from the default GC. Concepts like generations, segments, allocators, and finalization may not be available in a custom GC.

The simplest possible implementation of a Local GC is not very complex. It requires including only a few files directly from CoreCLR code to have things compiled: debugmacros.h, gcenv.base.h, and gcinterface.h. Please note that for brevity only most illustrative parts of such code is presented here. Refer to the accompanying book’s source repository for the whole, working example.

A custom GC library needs to define only two required exported functions, called by the CoreCLR during initialization: GC\_Initialize and GC\_VersionInfo (see Listing [15-37](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC37)). The former should specify custom implementations of two crucial interfaces: IGCHeap and IGCHandleManager. The latter is used to manage backward compatibility, as you can specify which version of runtime (its GC interface, more precisely) is required for our custom GC.

extern "C" DLLEXPORT HRESULT

GC\_Initialize(

/\* In \*/ IGCToCLR\* clrToGC,

/\* Out \*/ IGCHeap\*\* gcHeap,

/\* Out \*/ IGCHandleManager\*\* gcHandleManager,

/\* Out \*/ GcDacVars\* gcDacVars

)

{

IGCHeap\* heap = new ZeroGCHeap(clrToGC);

IGCHandleManager\* handleManager = new ZeroGCHandleManager();

\*gcHeap = heap;

\*gcHandleManager = handleManager;

return S\_OK;

}

extern "C" DLLEXPORT void

GC\_VersionInfo(

/\* Out \*/ VersionInfo\* result

)

{

result->MajorVersion = GC\_INTERFACE\_MAJOR\_VERSION;

result->MinorVersion = GC\_INTERFACE\_MINOR\_VERSION;

result->BuildVersion = 0;

result->Name = "Zero GC";

}

***Listing 15-37***Two required exported functions in Local GC library

We should additionally store the provided IGCToCLR interface address, used to communicate with CLR from inside our GC code. It contains a lot of methods and some of the most interesting ones are:

* SuspendEE and RestartEE - asks the runtime to suspend and resume managed threads, for a given reason (we can use it to implement not-concurrent parts of our custom GC).
* GcScanRoots - performs a stack walk of all managed threads and invokes the given promote\_func on all GC roots encountered on the stack (we would need this in our custom Mark phase implementation).
* GcStartWork and GcDone - inform the runtime that a GC has started and completed.

Custom IGCHeap interface implementation is the main interface representing core Garbage Collection functionality (see Listing [15-38](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC38)). Implementing IGCHeap requires implementing about 71 methods! Not all really need to have valid implementation though, as they are declared in built-in current GC design in mind - so we will provide some dummy implementations of methods like SetGcLatencyMode or SetLOHCompactionMode as our custom GC may does not have the concept of latency mode or LOH at all.

class ZeroGCHeap : public IGCHeap

{

private:

IGCToCLR\* gcToCLR;

public:

ZeroGCHeap(IGCToCLR\* gcToCLR)

{

this->gcToCLR = gcToCLR;

}

// Inherited via IGCHeap

...

}

***Listing 15-38***Fragment of custom IGCHeap implementation

Among various IGCHeap methods, the top-level methods are for allocations (IGCHeap::Alloc) and garbage collection (IGCHeap::GarbageCollect). The simplest possible so-called *Zero GC* (only capable of allocating objects but never reclaiming memory) could be implemented as in Listing [15-39](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#PC39). Please note that our custom GC does not have to distinguish “small” or “large” objects (and thus, SOH and LOH). We may allocate our objects as we wish regardless of its size - for example, by always using Heap API with the regular calloc function call.

class ObjHeader

{

private:

#ifdef \_WIN64

DWORD m\_alignpad;

#endif // \_WIN64

DWORD m\_SyncBlockValue;

};

Object \* ZeroGCHeap::Alloc(gc\_alloc\_context \* acontext, size\_t size, uint32\_t flags)

{

int sizeWithHeader = size + sizeof(ObjHeader);

ObjHeader\* address = (ObjHeader\*)calloc(sizeWithHeader, sizeof(char\*));

return (Object\*)(address + 1);

}

HRESULT ZeroGCHeap::GarbageCollect(int generation, bool low\_memory\_p, int mode)

{

return NOERROR;

}

***Listing 15-39***Examples of the 2 top-level methods implementation of the custom IGCHeap

It is really funny to see a single line of GarbageCollect method - the one that in case of default .NET GC triggers executing several thousand lines of code, described in hundreds of pages in this book. Here is where only our imagination is the limit. Feel free to implement your own GC!

By writing our custom GC, we replace all default GC functionality. Hence, it is not easy to just modify the default behavior “a little.” Although, if one takes the whole built-in GC code and will publish it as a Standalone GC library, it will be much easier to complete.

As write barriers are simply specially handled functions written in assembly code and injected by JIT, currently there is no API to replace them. As we may remember from Chapter [5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_5_Chapter.xhtml), write barriers are responsible for updating card tables so they are expected to exist, even if our implementation does not need them. Look for ZeroGCHeap::Initialize method in the accompanying example to see how IGCToCLR::StompWriteBarrier is configured to omit its usage by manipulating the lowest and the highest ephemeral segment address. And even if in custom GC, distinguishing between Workstation and Server mode should not make sense, because of write barriers, it still does matter: only in Workstation mode write barrier checks’ ephemeral segment boundaries (as explained in Chapter [5](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_5_Chapter.xhtml) in Listing [5-8](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_5_Chapter.xhtml#PC8)), so we can use it to omit card table updating. However, Server GC mode with our custom GC crashes the runtime because JIT\_WriteBarrier\_SVR64 is being used, which requires unconditionally valid card table address.

Please note that IGCHandleManager and IGCHandleStore dummy implementations are omitted for brevity. I invite you to read the Zero GC implementation provided with this book to see their code.

## **Summary**

This chapter described various ways of controlling and monitoring .NET memory usage programmatically. Based on the knowledge acquired from previous chapters, we should feel quite comfortable in writing code utilizing shown capabilities. As we might notice, knowledge about CLR and GC internals is quite often helpful, if not necessary, to properly configure and interpret data provided by libraries described in this chapter.

Firstly, comprehensive list of static GC class methods and properties was presented to summarize its already shown possibilities altogether with things that were not described well or not at all so far (like GC notifications). GC class usage was quite frequent throughout the book, so you’ve probably already noticed how useful it may be in various scenarios. From all the techniques described in this chapter, GC class (and a few auxiliary classes) seem to be the most common ones in an everyday’s developer work.

Then, CLR Hosting was presented with the most relevant interfaces on the field of memory management, to show what may be achieved with it. I do not expect big popularity of CLR hosting in your development, but I really wanted to present it to widen your toolbox. Maybe your use cases include calling managed code from unmanaged applications (like .NET scripting capabilities in SQL Server), so a possibility to manipulate how hosted CLR uses memory may be beneficial for you (with some monitoring capabilities available).

Presented ClrMD and EventTrace are two great libraries dedicated to deep diagnostic and monitoring of your .NET processes (including your own process in case of a self-monitoring scenario). Used together or alone, they allow us to get very detailed information about .NET runtime and your application’s behavior. Even they are overwhelmingly popular in implementing various diagnostic tools, you may also consider using it in self-monitoring scenarios as they provide relatively small overhead (a possibility especially tempting on pre-production environments).

Just in case you might be curious, the last section of this chapter presents a new possibility currently implemented only in .NET Core 2.1, which allows for a complete replacement of the GC implementation. I believe it greatly and ironically concludes the whole book, dedicated solely to the description of the default, built-in GC that may now be removed and replaced with something totally different. I strongly invite you to experiment with the Zero GC included as a sample of such custom GC. With the whole knowledge you’ve gained in this book, including theoretical introduction in the first chapters, you should now have the solid basics to start writing your own, not-so-trivial GC implementation!

**Footnotes**

[1](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fn1_source)

Strictly speaking, since there could be any number of things that happen between explicitly triggering a GC and calling GetTotalMemory method, some objects could also have become unreachable, unless there’s no other threads running.

[2](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fn2_source)

To use those properties, we have to enable Application Domain Resource Monitoring - refer to MSDN for ways of doing that.

[3](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fn3_source)

We should get used to numbering COM interfaces as it is a canonical way of taking care of backward compatibility. Instead of modifying an existing interface, a new one is added with an increased number.

[4](https://learning.oreilly.com/library/view/pro-net-memory/9781484240274/html/430794_1_En_15_Chapter.xhtml#Fn4_source)

For brevity, only the most relevant parts of code are presented in the subsequent examples. Refer to the accompanying GitHub repository to get full, working examples.

### **INDEX**

### **A**

AccessViolationException

Accumulator

Address lines

Address windowing extensions (AWE)

Allocation budget

AllocSmall

begin size

byte[] array

changes

C# program

ephemeral generations

ETW

GC events table

GCStats report

generation 0

generation 1

generation 2

generation sizes

gen0 survival rate

LOH budget

new allocation values

non-concurrent full-GC

OutOfSpaceSOH

per generation GC events

promotion size

SOH allocations

static GC Data

subsequent GCs

survival ratios

third GC

Visual Studio

Allocator

Amdahl’s law

AMD CodeAnalyst Performance Analyzer

Anscombe’s quartet

API, GC

explicit collection

finalization management

InternalCall

MemoryFailPoint usage

no GC region

notifications

properties and methods

GC.CollectionCount(Int32)

GC.GetAllocatedBytesForCurrentThread

GC.GetGeneration

GC.GetTotalMemory

GC.KeepAlive

GC.MaxGeneration

GCSettings.IsServerGC

GCSettings.LargeObjectHeapCompactionMode

GCSettings.LatencyMode

unmanaged memory pressure

AppDynamics

Application domains (AppDomains)

default domain

dynamic domains

.NET core

Shared Domain

System Domain

Application Performance Management (APM) tools

Arithmetic and logic unit (ALU)

Assemblies

DLL/EXE file

ORM

plugins

scripting

serializers

Assembly code

Async pinned handle

Automatic Computer Engine (ACE)

Automatic layout, classes

Automatic Sequence Controlled Calculator

### **B**

BenchmarkDotNet

Big data

Bimodal distribution

Binary search tree (BST)

Blittable types

Boxing

Brick table

### **C**

Cache coherency

Cache-coherency protocols

Cache hit

Cache lines

Cache miss

Caching

Call tree

Card bundles

Card tables

barrier code implementations

card word

data structure

JIT\_WriteBarrier function

JIT\_WriteBarrier\_PostGrow64 function

older-to-younger cross-generational references

organization in .NET runtime

set card

trade-off

Classic string concatenation

Clojure

CLR hosting

API

CLR memory usage data

configuration

configuration capabilities

customization

diagnostic measurements

executing code

heap-allocation implementation

host controller

IHostControl implementation

initialization

interfaces

ICLRAppDomainResourceMonitor

ICLRControl

ICLRGCManager2

IHostControl

IHostGCManager

IHostMalloc

IHostMemoryManager

memory management

memory-related interfaces

process

SetGCStartupLimitsEx

VirtualAlloc method

ClrMD API

AppDomains and modules

AttachToProcess

ClrRuntime instance

DataTarget class instance

documentation and samples

GC segments, process

internal data structures

LoadCrashDump

managed heap

managed processes and memory dumps

managed type instances

memory regions

reflection

thread’s call stacks

Collector

Commercial tools

Dynatrace and AppDynamics

Intel VTune Amplifier and AMD CodeAnalyst Performance Analyzer

JetBrains DotMemory

RedGate tool

Scitech .NET Memory Profiler

Visual Studio

Common Business Language (COBOL)

Common Intermediate Language (CIL)

Common Language Infrastructure (CLI)

Common language runtime (CLR)

exception handling

JIT compiler

memory management

type system

Common Trace Format (CTF)

Compacting process

Compaction

Compact phase

LOH

compacting objects

fragmentation ( *see* LOH fragmentation)

single loop scanning

SOH

compacting objects

copying objects

ephemeral segment

free-list items

generation boundaries

managed heap

memcopy function

overwrite

plan phase

plug info

relocate references

segments

temporary buffer

Compilation

Compiler

Computer architecture

Condemned generation

allocation budget

ephemeral generation

ephemeral segment

fragmentation limit

fragmentation ratio

fragmentation threshold

fragmented ephemeral

fragmented gen2

GCStats report

golden rule

induced explicit GC calls

internal tuning

latency mode

memory load

OutOfMemoryException

time-based tuning

workstation GC mode

ConditionalWeakTable class

Constrained execution region (CER)

CoreCLR, Linux environment

LTTng

mechanisms

memory dumps

monitoring and tracking applications

Perfcollect

perf\_events

Trace Compass ( *see* Eclipse Trace Compass)

tracking mechanisms, Windows and Linux

Core dump (memory dump)

CPU groups

CPU-Z

Crash dump

CreateMalloc method

Critical finalizers

Cross-generational references

### **D**

Dangling pointer

Data analysis

Data collection

Data-oriented design

designing types and data

software

strategic design

array of value-type nodes

data-oriented design

entity component system ( *see* Entity component system)

flattened tree

reference type array

repository of customers

structure-of-arrays

tree with nodes implementation

value-type array

tactical design

DebugDiag tool

Default Domain

Demotion event

Dependent handles

behavior

ConditionalWeakTable

!finalizequeue SOS extension command

GCHandle API

!gchandles SOS extension command

TryGetValue method

WeakEventManager class

Direct Memory Access (DMA)

Disposable objects

approaches

explicit cleanup

FileWrapper

GC.SuppressFinalize method

IDisposable interface declaration

IDisposable pattern

methods

System.Reflection library, CriticalDisposableObject

using clause

Dynamic domains

Dynamic memory allocation

Dynamic Random Access Memory (DRAM)

Dynatrace tool

### **E**

Eclipse Trace Compass

CoreCLR.GC.collections

CoreCLR.GC.generations.ranges

CoreCLR.threads.state

CTF format

final results

opening file

Entity component system

definitions

abstract system base

components

entity

inheritance tree

manager storing list

moving system

overview

Escape analysis

Event Tracing for Windows (ETW)

application crash

attributes

building blocks

consumer

controller

CPU-sampling

events and data

event tracking, Windows internals

kernel and user events

logman utility

CoreCLR

.NET-related ETW providers

manifest file

mechanism

MSDN documentation

.NET ETW providers

NT Kernel Logger session

PerfView ( *see* PerfView tool)

provider

session

Windows Performance Toolkit

Windows Performance Analyzer ( *see* Windows Performance Analyzer)

Windows Performance Recorder

Explicit allocations

ArrayPool

ArrayPool<T>.Shared instance

buckets

BufferAllocated event

ETW events

IArrayPool interface

Json.NET library

rent method

results

reusage ratio

structs

System.Buffers.ArrayPoolEventSource

object pool

memory zeroing

ObjectPool class

RecyclableMemoryStream

ETW events

LOH allocations

memory copying

memory waste

System.IO.Pipelines API

XML serialization

XmlWriter

stackalloc

LINQ

RuntimeHelpers.EnsureSufficientExecutionStack() methods

StackOverflowException

structs

structs

tuple

anonymous type

deconstruction

value tuple

ValueTask

async keyword

asynchronous method

AsyncTaskMethodBuilderstring struct

C# 7.0

IValueTaskSource interface

PooledValueTaskSource

ReadFileAsync method

returning async method

SetResult method

struct

task object

trade-offs

Expression stack

### **F**

Fibers

Finalization

avoid finalizers

BenchmarkDotNet benchmarks

caveats

critical finalizers

destructor

disadvantages

eager root collection

HandleCollector class

HandleRef struct

fill pointers

finalizable/non-finalizable objects

finalization array

finalizer thread

fReachable queue

GC.ReRegisterForFinalize(object) method

GC.SuppressFinalize(object) method

GC.WaitForPendingFinalizers method

lifetime logging example

limitations

memory leak

content of fReachable queue

finalization-related ETW events

!finalizequeue-allReady

!finalizequeue SOS command

finq and frq commands, SOSEX

fReachable queue

performance counters

memory pressure

queue

registering for

resurrection

example of

Simplified Timer class

safety nets

scenarios

slow-allocation path

user-defined code

Financial software

FlushAsync method

Fragmentation

threshold

virtual memory

Free Store

Front side bus (FSB)

FrugalObjectList<T>

FrugalStructList<T>

### **G**

Garbage collection

allocation budget ( *see* Allocation budget)

collection triggers ( *see* Triggers)

compact phase ( *see* Compact phase)

concurrent mode

dynamic data

allocation budget

attributes

current\_generation\_size method

generation 0

new allocation attribute

parameters limit

survival rate

EE suspension

analyzing time

defined

cooperative mode

fully interruptible code

execution engine

GC info

GC suspension mechanism

partially interruptible code

preemptive mode

safe point

stop the world technique

SuspendThread function

thread resuming

thread suspension

generation condemn ( *see* Condemned generation)

large object heap

non-concurrent mode

server GC modes

small object heap

static data

address limit

attributes

balanced mode

CPU cache size

ephemeral segment size

GC latency level

latency mode

memory footprint mode

mark phase ( *see* Mark phase, garbage collection)

sweep phase

CoreCLR code

LOH

plan phase

SOH

workstation GC modes

Garbage Collector (GC)

analyzing usage

ETW events

GC events

GC rollup by generation

GCStats report

GC utilization

Gen column

generation sizes

performance counters

PerfView

web application

API ( *see* API, GC)

condemned generation

custom GC

ETW/LLTng events

execution engine

fragmentation

full-GC

generation 0

compact collection

sweep collection

generation 1

generation 2

Managed Heap

mark phase

overhead

performance counters

SOH segment

steps

GC flavors

benchmarking GC modes

advantage

application

average response times

background server GC

CPU overhead

GC configurations

GC overhead

GC pauses

HdrHistogram.NET library

load test

memory usage

percentiles

response times

testing

choosing GC modes

concurrent mode

GC mode descriptions ( *see* GC mode)

GC settings

etrace tool

StartupFlags field

StartupFlags enumeration

StartupFlags value

latency ( *see* Latency mode)

modes configuration

application-centric path

ASP.NET web applications

configuration knobs

GC-centric approach

ICLRRuntimeHost interface

.NET Core

.NET Framework

Visual Studio

.NET

non-concurrent mode

pause and overhead

application side

CPU cycles

GC side

.NET metrics

operating modes

pause times

performance monitor tool

relative GC time

server mode

workstation mode

GC.GetAllocatedBytesForCurrentThread method

GC.GetGeneration method

GC heap

GC.KeepAlive method

GC mode

background server

background full-GC

characteristics

CPU cores

ephemeral collections

full-GC

usage scenarios

background workstation GC

allocation limit

background full-GC

background GC code

characteristics

concurrent mark phase

concurrent sweep phase

ETW/LLTNg events

full-GC

phases

stop the world phase

usage

concurrent mark phase

background GC phases

concurrent marking

final marking

floating garbage

lost object

mark array

phase

revisit objects

tracing collector

write watch list

WriteWatch mechanism

concurrent sweep

foreground GCs

server non-concurrent

characteristics

GCHeapCount

managed heaps

mark stealing technique

threads

usage scenarios

workstation concurrent

characteristics

concurrent full-GC

full-GC

usage scenarios

workstation non-concurrent

characteristics

ETW/LLTNg events

full-blocking GC

illustration

usage scenarios

GC profiling

cache mechanisms

compact phase

CPU stacks

ETW CPU profiling

ETW session

PerfView

plan phase

Generational garbage collection

Generation sizes in time

charts, ETW and performance counters data

ETW-based generation sizes

load test execution

measurements

performance counters data

performance monitor tool

web application execution

GetTotalMemory method

Gigabyte seconds (GB-s)

### **H**

Hardware

central processing unit (CPU)

cache

cache hit and miss

cache implementation

data alignment

data locality

hierarchical cache

multicore hierarchical cache

non-temporal access

prefetching

computer architecture

DDR4 memory chip

memory

modern architecture

Heap

deallocation

dynamic memory allocation

fragmentation

stack and

Heap-allocated object

assembler code

byRef interior pointer

local ref byRef variable

dumpheap and gcroot SOS commands

fragments of assembler code

interior pointer

managed pointer *vs.* regular object reference

objects relationships

pass by reference scenario

plug tree traversal and scanning

ref local with interior pointer

WeakReference type

Heap-allocation handling

Heap API

Heap segments

Hidden allocations

Azure functions

boxing

constraint

generic method

sources

value type method

closures

capturing state

compiler optimizations

lambda expressions

local functions

delegate

LINQ anonymous types

LINQ delegates

LINQ enumerables

hidden iterator allocation

immutability

iterators

string filtering method

LINQ queries

parameters array

string concatenation

system.generic collections

yield return

High Frequency Heap

### **I**

IGCHandleManager interface

IGCHandleStore interface

IGCHeap interface implementation

IHostMemoryManager interface

IMemoryOwner<T>

BufferedWriter class

FlexibleBufferedWriter.FlushAsync method

FlexibleBufferedWriter.WriteToBuffer method

interface declaration

method

problematic ownership

type

Immutable types

Indexing movable fixed buffers

Infant mortality

Instruction pointer (IP)

Instruction set architecture (ISA)

Intelligent pointers

Intel VTune Amplifier

Interior pointer

Interior pointer interpretation

Intermediate Language (IL)

Internal memory cells

### **J**

Java Virtual Machine (JVM)

JetBrains DotMemory

JIT\_GetSharedGCThreadStaticBase method

JIT\_GetSharedNonGCThreadStaticBase method

JIT\_WriteBarrier function

Just-in-time compiler (JIT compiler)

### **K**

Kernel space

### **L**

LargeHeapHandleTable structure

LargeObjectHeap

Large object heap (LOH)

array size

arrays of double

bump pointer technique

free-list allocation

gcAllowVeryLargeObjects setting

LargeHeapHandleTable structure

arrays

CoreCLR

Object[] arrays

RuntimeType

SOS extension

use

WinDbg

loh\_try\_fit method

OutOfMemoryException

slow path

sweeping GC

zeroing memory

layout of objects

plug information

result

Last in, first out (LIFO)

Latency mode

batch mode

CER

configuration knobs

enumeration

interactive mode

latency optimization goals

low-latency

no GC region

creation

GC.EndNoGCRegion method

GC.TryStartNoGCRegion method

sustained low latency

Latency to access memory

Latency *vs.* throughput

Lexical scope

*vs.* live stack roots

Lifetime partitioning

absolute time

card bundles

card tables ( *see* Card tables)

definition

generational GCs

generations

copying GC

logical boundaries

sizes measurements

relative time

remembered sets

CIL code

cross-generational references

generational GC

JIT\_WriteBarrier function

schematic pseudo-code, write barrier

write barriers in .NET

strong generational hypothesis

weak generational hypothesis

Little’s Law

Live debugging

Live stack roots

eager root collection

calling method

GC info

GC.KeepAlive method

memory usage

null settings

object behavior

optimizations

side effects

threads

Timer object

*vs.* lexical scope

Loader Heap

Local variable roots

fullPath

GC Info

calling methods

fully interruptible code

stack roots

untracked root

WinDbg, managed heap

lexical scope

live stack roots ( *see* Live stack roots)

pinned local variables

CIL code

fixed keyword

fragments of method

memory dumps

stack roots

stack root scanning

storage

LOH fragmentation

arrays

arrays pooling

blocking

callers view

!dumpheap command

!eeheap command

expected *vs.* observed size

!gcroot command

Gen 2 object deaths

holes

managed heap

performance counters

PerfView

processing code

server GC

SOS extension

strings

System.Byte[]

WinDbg

workstation GC

Long weak handles

Low Frequency Heap

### **M**

Machine learning (ML)

Managed pointers

C#-ref variables

limitations

object references

readonly ref variables and in parameters

ref locals

ref return

consuming ref returning method

limitations

local variable

null referencing reference

ref types internals

heap-allocated object ( *see* Heap-allocated object)

stack-allocated object

return type of methods

System.Int32 objects

types

Mark phase, garbage collection

finalization roots

GC handle roots

asynchronous I/O operation

!gchandles command

instance of normal object

managed objects

normal object

object with strong handle

pinned handles

pinned object

simple code

static object

string literal

strong handles

GC internal roots

local variable roots ( *see* Local variable roots)

memory leak ( *see* Memory leak)

object traversal and marking

popular roots

MarkWithType

promoted sized

Mark stack

Memory allocation

Allocator.Allocate(amount) method

avoiding allocations

garbage collection

memory optimization

premature optimization

sources

zeroing memory

bump pointer

allocate method

allocation context

allocation limit

allocation pointer

allocation quantum

ephemeral segment structure

fallback mechanism

garbage collection

infinite memory

multiple allocation contexts

sequential algorithm

simple sequential allocator

thread affinity

TLS

zeroing memory

explicit allocations, reference types ( *see* Explicit allocations)

free-list

best-fit

buckets

first-fit

free object

memory zeroing

unlinking

GC Managed Heap

Heap API

heap balancing

hidden allocations ( *see* Hidden allocations)

LOH ( *see* Large object heap (LOH))

object creation

allocation helpers

CIL

decision tree

JIT compilation

OutOfMemoryException ( *see* OutOfMemoryException)

SOH ( *see* Small Object Heap (SOH))

stack allocation

localloc CIL instruction

Span<T> type

stackalloc operator

StackOverflowException

unmanaged\_type

Memory bus

Memory dump

Memory leak

diagnostic

GCStats view

gen2 GCs

investigation

memory usage

performance counters

strings

Memory management

automatic

Allocator

Collector

garbage collection

LISP

memory handling

Mutator

reference counting

manual

ALGOL

characteristics

C program

C++ program

dangling pointer

free function

improvements

problems

.NET

Memory modules

Memory partitioning strategy

MemoryPool<T> class

Memory-related terms

address

assembly code

Automatic Sequence Controlled Calculator

binary code

binary number

bit

byte

control unit

Harvard architecture

heap

pointer

register

register machine

stack

activation frame

ALGOL

allocation

BURY and UNBURY

first compiler

FORTRAN

frame

LIFO

low-level mechanisms

machine

pop and push

StackOverflowException

Wheeler jump

von Neumann architecture

Williams tubes

word

Memory segments

Memory<T>

BufferedWriter class

explicit owner

internals

ReadOnlyMemory<T>

ReadOnlySpan<T>

rules

stack data

usages

Memory write watch mechanism

MESI protocol

Microarchitecture, processor

Modified Harvard Architecture

Mono

Mutator

### **N**

Named thread data slot

.NET memory dump

.NET Compact Framework

.NET Core

.NET Framework 1.0-4.7.2

.NET internals

CLR

exception handling

Execution Engine

GC

Hello World application

C#

execution

JIT compiler

JITted code, Main method

logical locations

CIL, Main method

SOS extension

WinDbg

managed code

memory-management mechanisms

misconceptions

.NET runtime execution

process

responsibilities

.NET memory management

call tree

core dump (memory dump)

description

invasiveness

latency *vs.* throughput

Linux environment ( *see* CoreCLR, Linux environment)

live debugging

measure early

measuring GC parameters

monitoring

objects graphs

dependency subgraph

retained size

retained subgraph

shallow size

shortest root path

total size

operating system

overhead

performance

sampling

statistics

Anscombe’s quartet

bimodal distribution

median, percentile and histogram

multimodal distribution

normal distribution

tools

Linux

Windows

tracing

Windows environment ( *see* Windows environment, .NET)

.NET Micro Framework

.NET Native

.NET versions

CLI

Mono

.NET Compact Framework

.NET Core

.NET Framework 1.0-4.7.2

.NET Micro Framework

.NET Native

Shared Source CLI

Silverlight

Windows Phone 7.x, 8.x, and 10 Mobile

WinRT

Non-nullable reference type

array, compiler behavior

class

compiler behavior

Non-uniform access memory

Non-uniform memory architecture (NUMA)

Non-unmanaged struct

Nullable reference types

class

compiler behavior

managed code generation

test method

unsafe code generation

Null Garbage Collector

NullReferenceException

### **O**

Object layout

ObjectLayoutInspector library

Object lifetime

description

disposable patterns ( *see* Disposable objects)

explicit cleanup

finalization ( *see* Finalization)

mechanisms

resource life cycle

SafeHandle

weak references ( *see* Weak handles)

Object-oriented programming

Object-relational mapping (ORM)

Objects’ layout

Operating system (OS)

large pages

Linux

memory layout

memory management

memory layout

memory manager

virtual memory

virtual memory fragmentation

Windows

memory layout

memory management

OutOfMemoryException

clrstack command

DebugDiag rule

GC

LOH

memory congestion

memory dump

objects allocation

objects pooling

physical backing store

regedit tool

virtual memory

VM hoarding

WinDbg

### **P, Q**

Passing by reference semantics

reference-type instance

value-type instance

Perfcollect script

Performance counters

advantage and disadvantage

application pools

architecture

attributes

counters

CPU usage

ETW data

garbage collection

instance process

monitoring tools

.NET CLR Memory category

Performance Monitor

Add Counters context option

Add Counters dialog

long-term analysis

parameters

short-term analysis

sample data

PerfView tool

configure symbol paths

data analysis

Any stacks view

GCStats view

Generic Events panel

sample ETL file

data collection

description

main actions

memory snapshots

options

startup

tabular view

Physical partitioning

allocation patterns

blocks and segments

default segment sizes

heap segments

large object heap waste

managed heaps

segments and generations information, WinDbg

segments and heap anatomy

segments reuse

segment types

server mode

single block, ASP.NET application

workstation GC initial segments configuration

workstation mode

Pipelines

AdvanceTo method

API

buffered memory

characteristics

configuration

FlushAsync method

GetSpan and GetMemory methods

KestrelHttpServer

ParseRequestLine

Reader.Buffer

usage

zero-copy read side

Plan phase

LOH ( *see* Large object heap (LOH))

SOH ( *see* Small Object Heap (SOH))

Plugins

Pointer

Pointer arithmetic

Post-mortem analysis

ProcDump tool

Process memory regions

dumpbin command-line program

heap and private data

images

Managed Heaps

domains heaps

GC Heap

mapped files

measurements

private bytes

private working set

virtual bytes

.NET runtime

page tables

Performance Monitor tool

program’s memory usage

AssemblyLoad events

commercial tools

gigabytes of memory

growth of private bytes

Lookup Symbols

Managed Heap size

memory leak

MSDN documentation, XmlSerializer

Net OS Heap Alloc Stacks

Performance Counter

Performance Monitor tool

PerfView tool

Task Manager

VirtualAlloc call

VMMap

RAM

shareable

stacks

unusable

VMMap tool

Window’s Task Manager

Processor groups

Program counter (PC)

### **R**

Random Access Memory (RAM)

Reachability of object

Read-only heap segments

ReadOnlyMemory<T>

Readonly struct

RedGate ANTS Memory Profiler

Reference counting

advantages

circular references

C++ program

dangling pointer

data.use\_count() method

disadvantages

exception handling

liveness of objects

Mutators

pseudo-code

smart pointers

Reference types

classes

definition

heap allocated object

memory layout

method table reference

object header

sample code

local variable sd

entities

escape analysis

heap allocation possibilities

object type

pointer type

Ref structs

Register machine

Remembered sets

Resource Acquisition Is Initialization (RAII)

Resource life cycle *vs.* object

Roots

Rotor

### **S**

SafeHandle object

advantages

DangerousGetHandle method

fragments of

handle-recycling attack

implementation

IntPtr

IsInvalid and ReleaseHandle

P/Invoke calls

resources

System.Runtime.InteropServices.SafeHandle class

Sampling

Scitech .NET Memory Profiler

Segmentation fault error

Serializers

SGen Garbage Collector

Shared Domain

Shared Source CLI

Short weak handles

Silverlight

Simultaneous multithreading mechanism (SMT)

Size partitioning

LargeHeapHandleTable

LOH ( *see* Large object heap (LOH))

small object heap

SmallObjectHeap

Small Object Heap (SOH)

brick table

demotion

fastest allocation helper

generation boundaries

heavy-allocating library

investigating pinning

ETW-based session

local pinned variables

pinned handles

memory dump

OutOfMemoryException

pinned object

pinned plug

after gap

implications

before marked objects

normal plug

plug tree

queue

plugs and gaps

BST

Managed Heap

relocation offset

size and offset information

pointer technique

slow path

soh\_try\_fit() method

Smart pointers

SOLID principles

Span <T>

compiler

Fast Span

internals

ReadOnlySpan<T>

rules

simplified int parsing API

Slow Span

usage

concise conditional local buffer acquiring

OnStartLine method

scenarios

ValueStringBuilder

Spatial locality

Stack-allocated object

Stack roots

Static data

internals

implementation

JIT-compiled code

JIT compiler

Object[] array

primitive static field

reference-type

storage in .NET Core

types

user-defined value type

static fields

Static memory allocation

Static Random Access Memory (SRAM)

Stored-program computers

StringBuilderCache class

StringBuilder instance caching

StringFreezingAttribute class

String interning

advantages

code

disadvantages

duplication analysis, JetBrains dotMemory tool

internals

JIT compilation

manual

optimization technique, repetitive texts

PerfView graph, allocation

string duplication

string.Intern method

String Literal Map

Strings

benchmark results

concatenation and hidden temporary string creation

Concat method

design decisions

FormatHelper method

Greet method

immutability

interning ( *see* String interning)

mutable string

reference type

StringBuilder

StringBuilderCache

StructLayout attribute

Structs

advantages

arrays

automatic field’s layout

avoid allocation

boxing

default fields layout

definition

discriminated union

explicit field’s layout

field layout

fixed size buffer

LayoutKind.Auto layout

memory layout

memory region

ObjectLayoutInspector

readonly

ref structs (byref-like types)

sequential layout

Sharplab.io, memory layout

storage

CIL code of Main method

evaluation stack

Helper method

JIT compiler

local variables

locations

sample code

SomeData

unmanaged type

Stub Heap

SuperBenchmarker

Symmetric multiprocessing (SMP)

System Domain

### **T**

Tactical data-oriented design

Tactical design

cache levels

cache line utilization

LayoutKind.Automatic

parallel processing

random memory access

ThreeItemList<T> class

Temporal locality

Thread affinity

Thread data slots

ThreadLocalInfo structure

ThreadLocalModule

Thread local storage (TLS)

definition in CoreCLR

internals

CLR internal data

generic types

Object[] arrays and static blobs

structs

thread affinity

ThreadLocalBlock

ThreadLocalInfo structure

ThreadLocalModules

thread static data

ThreadStaticHandleTable

type thread-static fields

multithreading synchronization techniques

performance advantages

thread data slots

thread static fields ( *see* Thread static fields)

usage scenarios

ThreadLocal<T> class

Thread-specific data

Thread static fields

field initialization

initialization, regular static field

primitive and reference TLS

SomeClass

SomeOtherClass.Run method

ThreadLocal<T> usage

value and reference types

Value property

ThreadStaticHandleTable

TraceEvent Library

Tracing GC

Collect phase

Compact

Sweep

Mark phase

conservative garbage collector

marking process

Precise GC

states of object

steps

Triggers

allocation

explicit

batch processing

benchmarking, cleaning

GC.AddMemoryPressure

GC.Collect method call

generation 0

generation 1

generation 2

memory usage

proactive cleaning

WeakReference

explicit GC calls

AddMemoryPressure method

bitmaps

dispose method

events view

IDisposable interface

manual memory cleaning

performance counter

reason field

SafeMILHandleMemoryPressure class

stack trace

internal triggers

low memory level system

memory usage

self-tuning GC

Types data locality

Type storage

Type system

identity

immutable types

implementation details

lifetime

memory-management

MethodTable

reference types ( *see* Reference types)

sharing

type storage

value types ( *see* Value types)

### **U**

Unboxing

Unmanaged constraint

blittable types

generic constraint usage

generic logging mechanism

generic serialization

object passed by reference

regular struct usage

struct method

type wrapping unmanaged memory

usage

Unmanaged type

Unnamed thread data slot

Unsafe internals

class API

class usage

Array.Reverse static method

BitConverter class

casting

jemalloc.NET library

MemoryMarshal helper class

static methods

MemoryMarshal usage

method implementation

methods

User space

### **V**

ValueStringBuilder class

Value types

definition

enumerations

storage

arguments of method

evaluation stack

instance field

local memory pool

local variables

static field

structs ( *see* Structs)

Virtual address space

Virtual API

Virtual Call Stub

Virtual memory

Virtual stub dispatching (VSD)

Visual Studio

VMMap tool

### **W, X, Y**

Weak handles

caching

Gen2GcCallback class

long weak handles

object type and members

observers and listeners

short weak handles

types

weak events

child windows

WeakEventManager class

Windows Presentation Foundation

WeakReference<T> type

WinDbg

commands

extensions

NetExt

SOS

SOSEX

installing

main window

msos tool

.NET runtime

operations

Windows 10 Mobile

Windows Driver Kit (WDK)

Windows environment, .NET

BenchmarkDotNet

commercial tools ( *see* Commercial tools)

DebugDiag

disassemblers and decompilers

ETW ( *see* Event Tracing for Windows (ETW))

performance counters ( *see* Performance counters)

PerfView ( *see* PerfView tool)

ProcDump

VMMap

Windows Performance Analyzer ( *see* Windows Performance Analyzer)

Windows Performance Recorder

Windows Performance Analyzer

custom graphs

description

flame charts

generic events

opening file and configuration

profiles

region of interests

stack tags

SuperBenchmarker

Windows Performance Recorder

Windows Phone 7.x

Windows Phone 8.x

WinRT

### **Z**

Zero Garbage Collector