Managed Heaps

To recap, when a .NET application runs, two sections of memory are reserved for storing objects allocated during execution (we'll ignore the other two sections of the heap for now). The memory sections are called the Small Object Heap (SOH) and the Large Object Heap (LOH). No prizes for guessing the difference! The SOH is used to store objects smaller than 85 K, and the LOH for all larger objects.

How big is an object?

I'll open this section by stating that memory is more complex than we'll be able to encapsulate in this chapter, but the picture I'll paint for you will be accurate as far as my explanations go and, more importantly, useful when it comes to helping you troubleshoot memory-related problems. It's tempting to assume an object's size on the heap includes everything it contains. In fact, it doesn't include the objects it, in turn, contains, as they are allocated separately on the heap.

Consider the following code:

```
class MyClass
{
    string Test="Hello world Wazzup!";
    byte[] data=new byte[86000];
}
```

Listing 2.1: Allocating `MyClass` .

It's easy to assume that the size of `MyClass` when allocated includes:

- 19 characters
- 86,000 bytes.

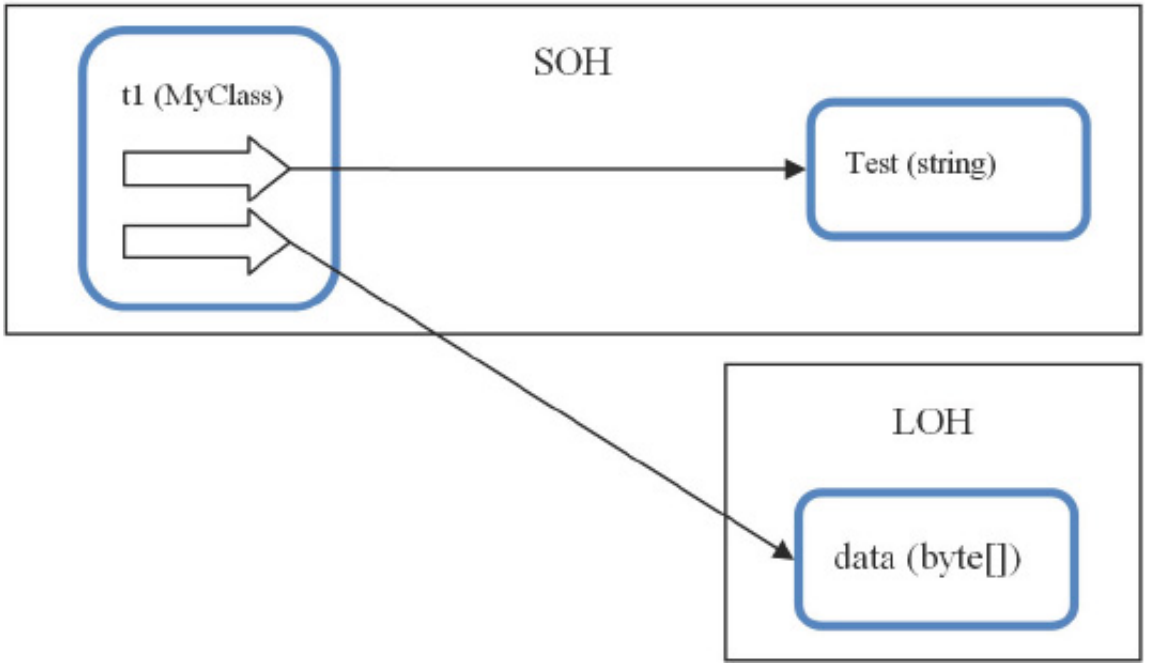

Figure 2.1: Heap allocation locations.

In fact, the object's size will only include general class stuff, and the memory required to store the object pointers to the string and the byte array (class level variables), which are then separately allocated onto the heaps. The string will be allocated on the SOH, and its object reference held by the instance of the class; the byte array will be allocated onto the LOH, as it's bigger than 85 KB.

In Chapter 1 we discussed how object references can be held for objects which are allocated onto one of the heaps from:

- other .NET objects
- the stack
- CPU registers
- statics/globals
- the finalization queue (more later)
- unmanaged Interop objects.

To be of any use to the application, an object needs to be accessible. For that to be the case, it either needs to have a reference pointing to it directly from a root reference (stack, statics, CPU registers, finalization queue), or it needs to be referenced from an object that ultimately has a root reference itself. All I mean by that is that if, by tracking back through an object's reference hierarchy, you ultimately reach a root reference, then all of the objects in that hierarchy are fundamentally accessible (rooted).

This simple principle is the cornerstone of how .NET memory management works. Using the rule that an object which is no longer accessible can be cleaned up, automatic garbage collection becomes possible.

What is automatic garbage collection?

Put simply, automatic garbage collection is just a bunch of code that runs periodically, looking for allocated objects that are no longer being used by the application. It frees developers from the responsibility of explicitly destroying objects they create, avoiding the problem of objects being left behind and building up as classic memory leaks.

The GC in .NET runs for both the LOH and SOH, but also works differently for both. In terms of similarities, each of the heaps can be expanded by adding segments (chunks of memory requested from the OS) when necessary. However, the GC tries to avoid this by making space where it can, clearing unused objects so that the space can be reused. This is much more efficient, and avoids expensive heap expansion.

When does the GC run?

The GC runs on a separate thread when certain memory conditions are reached (which we'll discuss in a little while) or when the application begins to run out of memory. The developer can also explicitly force the GC to run using the following line of code:

```
GC.Collect();
```

Listing 2.2: Forcing the GC to collect.

Can I just add that this is never really a good idea because it can cause performance and scalability problems. If you find yourself wanting to do this, it usually means things aren't going well. My advice is: take a deep breath, then get hold of a good memory profiler and find a solution that way.

Small Object Heap

Allocation and automatic garbage collection on the Small Object Heap (SOH) is quite a complex process. Because most of the objects allocated within an application are less than 85 K, the SOH is a pretty heavily used storage area. So, to maximize performance, various optimizations have been added which, rather unfortunately, add to the complexity.

To aid your understanding, I am going to describe the SOH at a relatively high level to start with, and then add the complexities slowly until you have the full picture. Trust me, you will thank me for it!

Consecutive allocation on the SOH

In unmanaged C/C++ applications, objects are allocated onto the heap wherever space can be found to accommodate them. When an object is destroyed by the programmer, the space that that object used on the heap is then available to allocate other objects onto. The problem is that, over time, the heap can become fragmented, with little spaces left over that aren't usually large enough to use effectively. As a result, the heap becomes larger than necessary, as more memory segments are added so that the heap can expand to accommodate new objects.

Another problem is that whenever an allocation takes place (which is often), it can take time to find a suitable gap in memory to use.

To minimize allocation time and almost eliminate heap fragmentation, .NET allocates objects consecutively, one on top of another, and keeps track of where to allocate the next object.

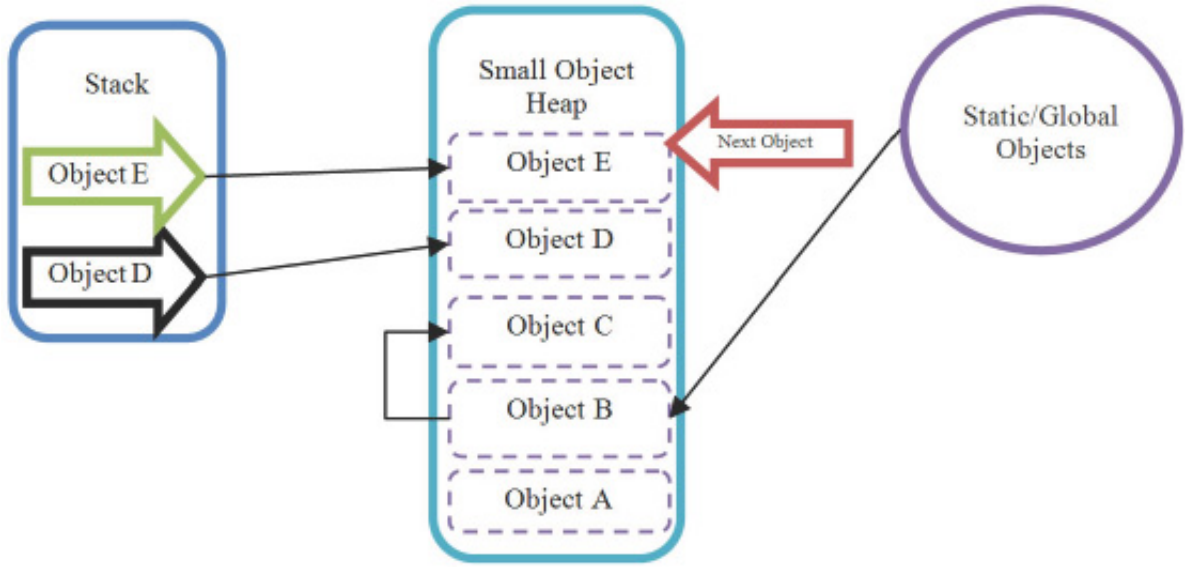

Figure 2.2: The Small Object Heap.

Figure 2.2 demonstrates how this works. You can see a number of objects allocated on the SOH, as well as the references that are held on the stack from other objects and from statics. In fact, all of the objects except Object A have a reference that can be traced back to a root.

Each of the objects are allocated on top of each other, and you can see there is an arrow which represents the location where the next object will be allocated; this represents what is known as the Next Object Pointer (NOP).

Because the NOP position is known, when a new object is allocated, it can be instantly added at that location without the additional overhead of looking for space in Free Space tables, as would happen in C/C++ unmanaged applications.

The only slight downside of this picture as it stands is that there is a potential for memory to become fragmented. Looking back at Figure 2.2, you can see that Object A has no references to it and it's at the bottom of the heap. As a result, it's a prime candidate to be garbage collected, as it is no longer accessible to the application, but it would leave a gap on the heap.

To overcome the fragmentation problem, the GC compacts the heap (as mentioned in the previous chapter), and thereby removes any gaps between objects. If the GC cleaned up the heap in Figure 2.2, it would look like the example in Figure 2.3 after the GC had completed its work.

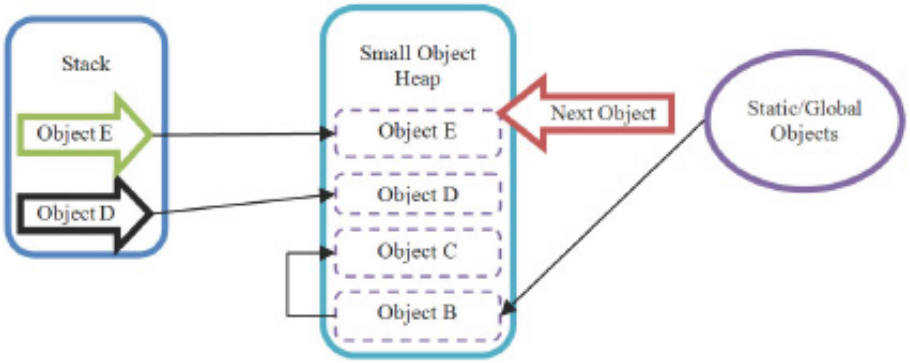

Figure 2.3: SOH after compaction.

Notice that Object A has been removed, and that its space has been taken up by the other objects. As a result of this process, fragmentation of the SOH is drastically reduced, although it can still happen if objects are deliberately pinned in memory (more on this in Chapter 3).

What's still in use?

In Chapter 1 we looked conceptually at the kind of algorithms used by the GC to determine which objects are still being used and those that aren't. The rule is really very simple: "If you don't ultimately have a root reference, then you can't be accessed, so you must die." Harsh, maybe, but very effective.

When it runs, the GC gets a list of all root references and, for each one, looks at every referenced object that stems from that root, and all the objects they contain recursively. Every object found is added to a list of "in use objects." Conversely, any object not on the list is assumed to be dead and available for collection, which simply involves compacting the heap by copying live objects over the top of dead ones. The result is a compacted heap with no gaps.

Conceptually, everything I just said is correct but, as I mentioned in Chapter 1, whether the actual GC code performs "object in use" processing with a separate marking phase, sweeping phase, and compacting phase, is sometimes difficult to determine. What is clear is that the GC algorithm will be optimized to the maximum for releasing objects as soon as possible, while touching as few objects as possible. While it doesn't always get it right, that's the context you should always see it in.

Optimizing garbage collection

If you think about it, there's potentially a bit of a problem with creating "still in use" lists and compacting the heap, especially if it's very large. Navigating through huge object graphs and copying lots of live objects over the top of dead ones is going to take a significant amount of processing time. To get maximum performance, that whole process needs to be optimized, and that's just what the .NET GC guys did.

They decided to classify all objects into one of three groups. At one end, you've got short-lived objects that are allocated, used and discarded quickly. At the other end of the spectrum, you have long-lived objects which are allocated early on and then remain in use indefinitely. Thirdly and finally, you have, you guessed it, medium-lived objects, which are somewhere in the middle.

When they analyzed typical allocation patterns they discovered that short-lived objects are far more common than long-lived ones. They also realized that the most likely contenders for GC were the most recently allocated objects. Because most objects are created within a method call and go out of scope when the method finishes, they are, by definition, short lived. So a really cool optimization was developed, which involved inspecting and collecting the most recently allocated objects more frequently than the older objects.

This was a simple and elegant approach that reduced the number of root references to be inspected and the number of object hierarchies to be navigated through. It also reduced the amount of heap compaction required.

The only slight problem lies with how they decided to name short-, medium- and long- lived objects: Generation 0 (Gen 0), Generation 1 (Gen 1), and Generation 2 (Gen 2), respectively. Obviously, they didn't consult the Marketing department first!

For all that they lack descriptive qualities, these terms are widely known and understood, and I will (only slightly begrudgingly) use them from now on, as I don't want to confuse you with my own personal preferences.

Generational garbage collection

You know the diagram I showed you in Figure 2.2? The one that was simple to grasp, and which you clearly understood? Well, let's now throw that away (I did warn you), because we now need to think of the SOH in terms of the three generations of objects: Gen 0, Gen 1, and Gen 2.

When an object has just been created, it is classified as a Gen 0 object, which just means that it's new and hasn't yet been inspected by the GC. Gen 1 objects have been inspected by the GC once and survived, and Gen 2 objects have survived two or more such inspections (don't forget that the GC only lets an object survive if it ultimately has a root reference).

So let's look at some allocated objects in their respective generations.

In Figure 2.4, you can see each of the objects on the SOH and the generations they belong to. Object W must have survived at least two GCs, whereas Z and Y have yet to be looked at for the first time. Object X has survived one GC inspection.

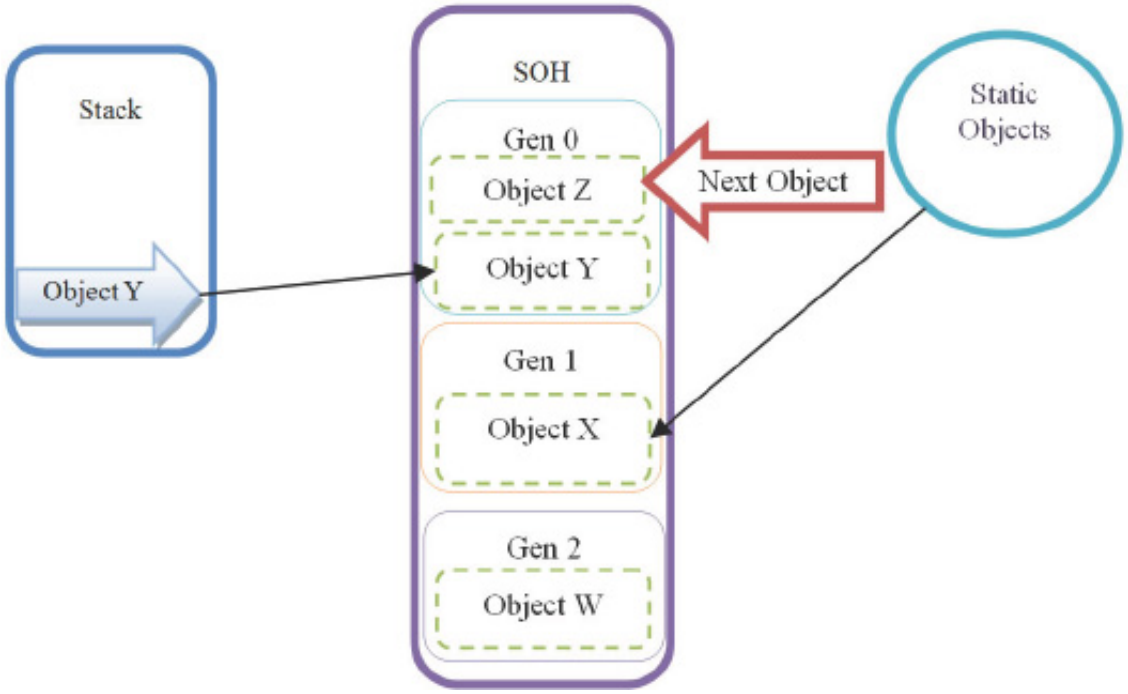

Figure 2.4: SOH with generations.

The GC runs automatically on a separate thread under one of the conditions below.

- When the size of objects in any generation reaches a generation-specific threshold. To be precise, when:
  - Gen 0 hits ~256 K
  - Gen 1 hits ~ 2 MB (at which point the GC collects Gen 1 and 0)
  - Gen 2 hits ~10 MB (at which point the GC collects Gen 2, 1 and 0)
- `GC.Collect()` is called in code
- the OS sends a low memory notification.

It's worth bearing in mind that the above thresholds are merely starting levels, because .NET modifies the levels depending on the application's behavior.

GC operation also depends on whether the application is server- or workstation-based, and on its latency mode. To avoid confusion at this stage, let's go through the general principles of generational garbage collection first, and then we will cover these more advanced topics in the next chapter.

So now let's take a look at what happens when a GC is triggered from each of the generations.

Gen 0 collection

Gen 0 objects have never been looked at by the GC before, so they have two possible options when they are finally inspected:

- move to Gen 1 (if they have a root reference in their hierarchy)
- die and be compacted (if they are rootless and therefore no longer in use).

Either way, the result of a Gen 0 GC is an empty Gen 0, with all rooted objects in Gen 0 being copied to (and reclassified as) Gen 1, joining the other Gen 1 objects. Remember, in a Gen 0 collection, existing Gen 1 objects stay where they are.

Figure 2.5 shows the state of the SOH after a GC on the heap (as previously displayed in Figure 2.4). Notice how Object Y has now joined Object X in Gen 1, leaving Gen 0 empty.

Object Z didn't have a root reference, and so will be effectively overwritten by the next allocation, because the NOP is simply repositioned below its heap location.

After a GC, Gen 0 will always be empty.

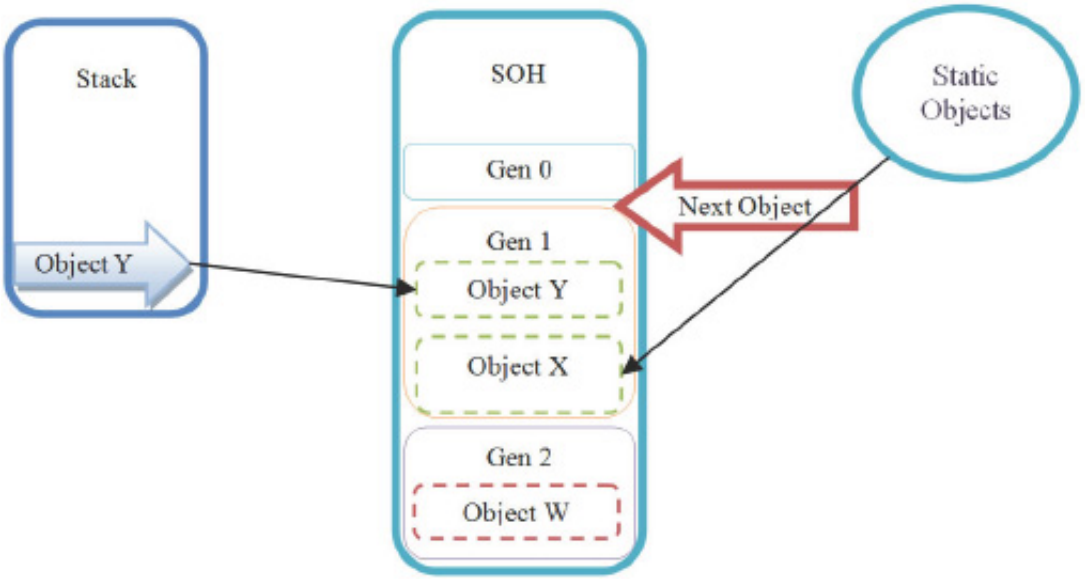

Figure 2.5: SOH after a Gen 0 collection.

Gen 1 collection

Gen 1 collections collect both Gen 1 and Gen 0 objects. Again, to illustrate, let's see the outcome of a Gen 1 collection on the heap state in Figure 2.4.

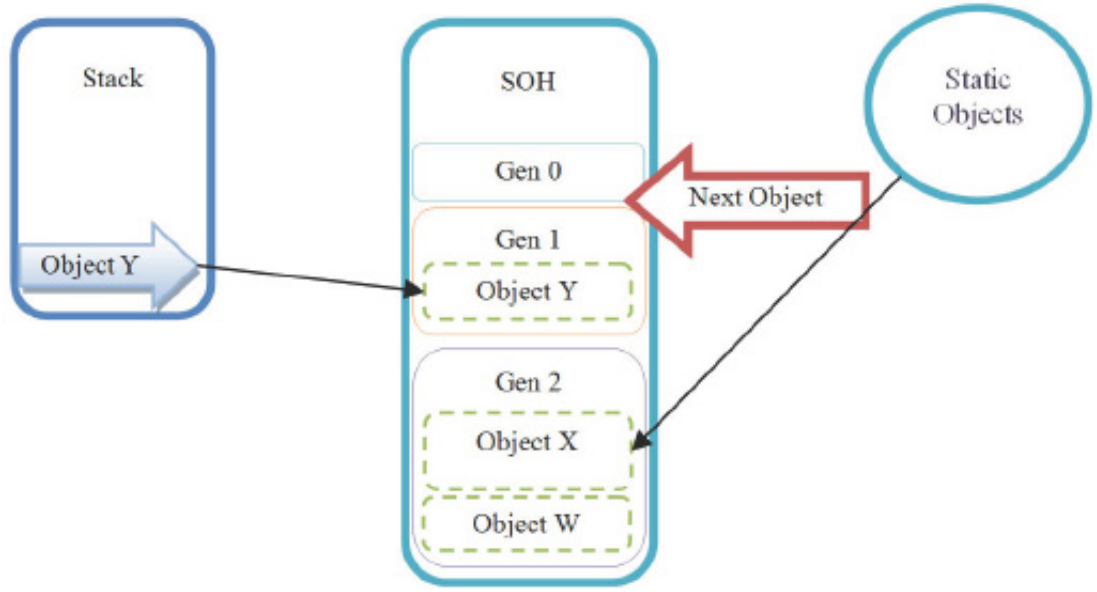

Figure 2.6: SOH after Gen 1 collection.

This time, instead of Object X staying in Gen 1, it is moved to Gen 2 (after all, it's now survived two GCs). Object Y moves from Gen 0 to Gen 1 as before and Object Z is collected. Once again, Gen 0 is left empty.

Gen 2 collection

Gen 2 collections are known as "Full" collections because all of the generations are inspected and collected. As a result, they cause the most work and are thus the most expensive.

Incidentally, if you've been worrying about why Object W has been surviving from Figure 2.4 all this time, then you can stop fretting, because it's about to die.

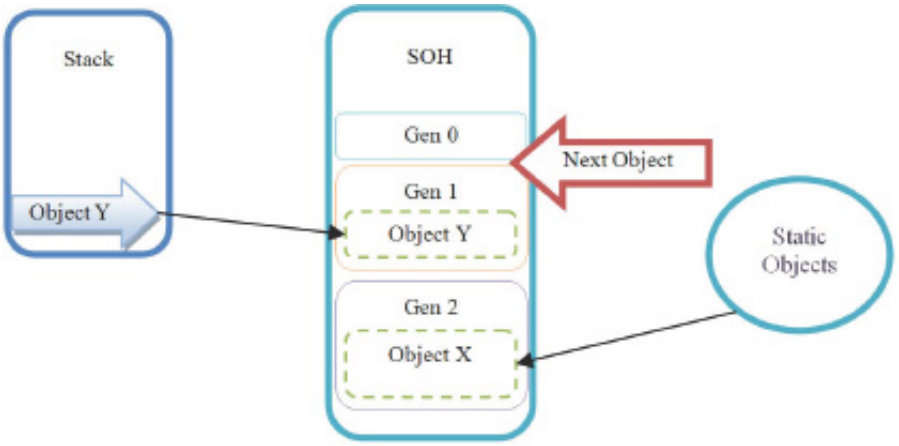

Figure 2.7: SOH Gen 2 collection.

As you can see in Figure 2.7, a full Gen 2 collection results in the following:

- Object W dies and is compacted
- Object X moves to Gen 2
- Object Y moves to Gen 1
- Object Z dies and is compacted.

Obviously, the GC has had to do more work in this instance than in either of the previous collections. Ideally, you only want objects to make it to Gen 2 if they absolutely need to be there. Gen 2 is an expensive place to keep your objects, and too frequent Gen 2 collections can really hit performance.

The general rule of thumb is that there should be ten times more Gen 0 collections than Gen 1, and ten times more Gen 1 collections than Gen 2.

Performance implications

You might be tempted to think, having learned all this, that the key to improving GC performance is creating as few objects as possible. This isn't quite right. True, the fewer objects you create, the fewer GCs you will incur; however, the key is more how many objects are removed, rather than how often the GC runs. By this I mean that, so long as your Gen 0 collections are removing a lot of objects (which is a relatively cheap process, remember), fewer objects are being promoted, and so the performance stays high.

So, in an ideal world, the vast majority of your objects should die in Gen 0, or possibly Gen 1. Objects that make it to Gen 2 should do so for a specific reason; probably because they're reusable objects that it is more efficient to create once and keep, rather than recreate each time.

If an object makes it to Gen 2, it's often (or should be) because it's actually needed. Unfortunately, you can, and will, write code where objects make it to Gen 2 only to then immediately lose their root reference. This means that they are not in use, but are potentially going to stick around and take up valuable space. In addition, if this is code that runs frequently during execution, it means that Gen 2 GC is going to happen a lot more frequently.

Finalization

If you write classes that access unmanaged resources such as files/disks, network resources, UI elements or databases, then you have probably done the decent thing and written cleanup code in your classes to close and destroy the used resources.

Any unmanaged resource accessed from .NET won't be cleaned up by the GC and so will be left behind. If this is news to you, then please feel free to put the book down right now, go and add some destructor code to the offending classes, and we'll say no more about it.

Joking aside, if you put a destructor or a `Finalize` method (henceforth known as the **finalizer**) in your class (see Listing 2.3), then you will actually extend the lifetime of instances of your classes for longer than you expect.

You can put your cleanup code in either method below and .NET will, at some point, call it before the object is destroyed, ensuring your object cleans up just before it is destroyed.

```
class TestClass
{
      ~TestClass()
      {
      }
}

class TestClass2
{
      void Finalize()
      {
      }
}
```

Listing 2.3: Destructor/finalizer code.

You would probably think that, when an object is ready to be collected, and if it's got a `Finalize` method, then the GC could just call that finalizer and then compact the object out of existence. But think about it; that's potentially asking the GC to be closing files and databases and all the other stuff you might put inside a destructor (depending on your view of the theory of infinite universes, there's a finalizer out there that downloads Tom Petty's Greatest Hits and uploads them to DropBox.) What that means is that it's potentially going to slow the GC down quite a bit.

So the guys at Microsoft took a different approach, and decided to call the finalizer on objects asynchronously and on a dedicated thread. This runs periodically and calls the finalizer on all applicable objects, entirely independently of the GC.

However, that creates a new puzzle: how do you prevent an object that needs finalization from being compacted before its finalizer is called? The answer they came up with was to keep a queue of extra root references to all finalizable objects (one reference per object) and use this to keep them alive long enough to call their finalizer.

Let's look at how it works.

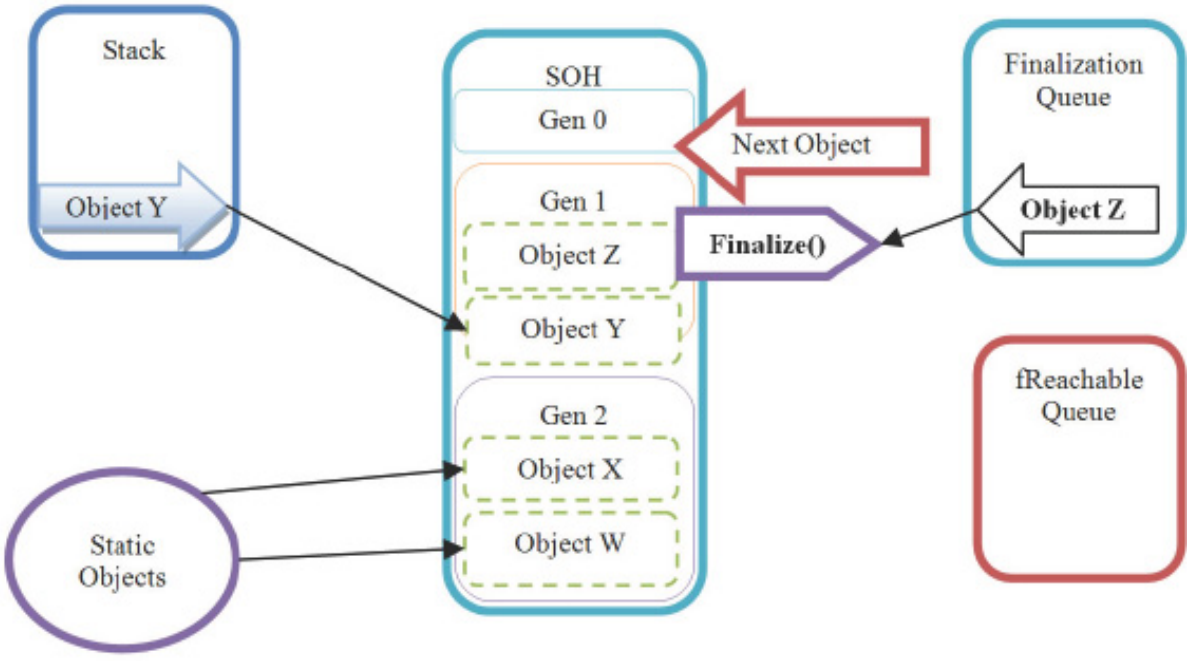

Figure 2.8: Finalization mechanism.

In Figure 2.8 you can see Object Z has a finalizer method on it. When it's created, as well as a root reference in one of the usual places (stack, statics, etc.), an additional reference is added onto the finalization queue, which then acts as a kind of reminder to .NET that it needs to call the finalizer on this object at some point.

When Object Z loses its root reference, it would usually become a candidate for collection but, because of the extra reference on the finalization queue, it's still viewed as “rooted,” and isn't collected when the GC next runs. Instead, it's promoted to Gen 2

(assuming a Gen 1 collection occurs).

Figure 2.9 shows Object Z being promoted from Gen 1 to Gen 2, and you should notice how the finalization reference is also moved from the finalization queue to another queue, called the fReachable queue. fReachable acts as a kind of reminder list of all the objects on the heap that still need to have their finalizer called. Think of it like this; the finalization queue keeps a reference to all of the live finalizable objects and fReachable references dead objects that need their finalizer calling.

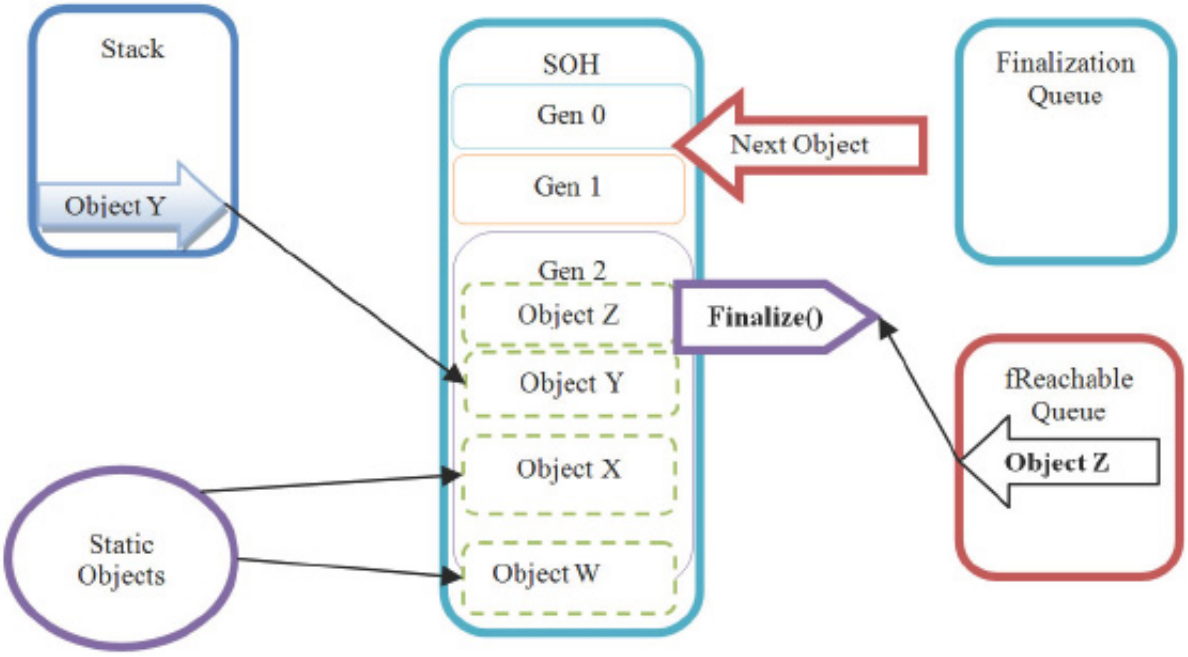

Figure 2.9: Promotion of a finalizable object to a later generation.

Periodically, the finalization thread will run, and it will iterate through all objects pointed to by references in the fReachable queue, calling the `Finalize` method or destructor on each one and removing its reference from fReachable. Only then will the finalizable object be rootless and available for collection.

In the earlier example, Object Z made it to Gen 2, where it could potentially have remained for a while. What we don't know is whether Object Z was actually needed for all that time, or whether its finalization was just poorly implemented.

This doesn't mean that finalizers are bad; in fact, they are absolutely essential. But you do need to write your finalization mechanism in the right way, which is what comes next.

Improving finalization efficiency

A simple pattern you can follow to avoid the finalization problem, is by implementing the `IDisposable` interface on your class and applying the following `Dispose` pattern:

```
public void Dispose()
{
     Cleanup(true);
     GC.SuppressFinalize(this);
}
private void Cleanup(bool disposing)
{
     if (!disposing)
{
     // Thread-specific code goes here
}

// Resource Cleanup goes here
```

```
}
public void Finalize() ;
{
     Cleanup(false);
```

Listing 2.4: Improving finalization efficiency using `Dispose`.

Listing 2.4 shows the standard `Dispose` pattern you can add to your finalizable class. All you do is provide a standard `Cleanup` method that is responsible for cleaning up the class. It needs a Boolean parameter, which is used to determine if `Cleanup` was called from the finalizer or directly from code. This is important because the finalizer is called on a separate thread, so if you are doing any thread-specific cleanup, then you need to avoid doing it if it's the finalizer running it.

Finally, we add a `Dispose` method which calls the `Cleanup` method, passing `true.` Crucially, it also calls `GC.SuppressFinalize(this)`, which deletes the reference in the finalization queue and gets around the problem.

The `Finalize` method also calls the `Cleanup` method but just passes `false`, so it can avoid executing any thread-specific code.

The developer can now use the class, and either explicitly call `Dispose`, or call it within a `using` statement (Listing 2.5). Both will ensure cleanup and avoid object lifetime extension.

```
// Explicitly calling Dispose
FinObj myFinObj=new FinObj();
myFinObj.DoSomething();
myFinObj.Dispose();

// Implicitly calling Dispose
using (FinObj myFinObj=new FinObj())
{
    myFinObj. DoSomething ();
}
```

Listing 2.5: Using the `Dispose` pattern to avoid object lifetime promotion.

Large Object Heap

Thus far we've been fairly focused on the SOH, but objects larger than 85 KB are allocated onto the Large Object Heap (LOH). Unlike the SOH, objects on the LOH aren't compacted, because of the overhead of copying large chunks of memory. When a full (Gen 2) GC takes place, the address ranges of any LOH objects not in use are recorded in a “free space” allocation table. If any adjacent objects are rootless, then they are recorded as one entry within a combined address range.

In Figure 2.10 there are two free spaces that were occupied by objects that have become rootless. When the full GC ran, their address ranges were simply recorded in the Free Space table.

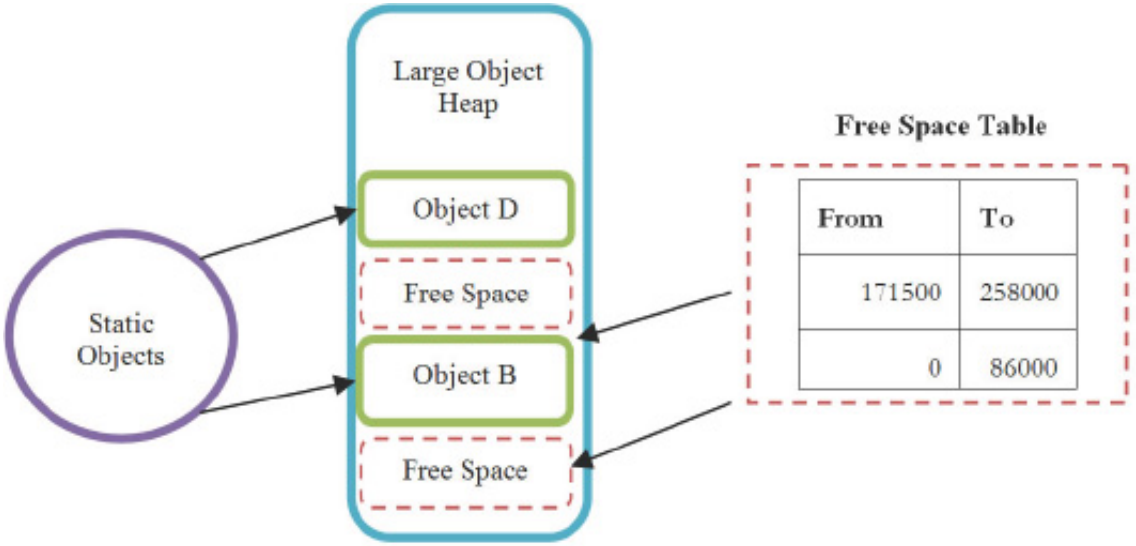

Figure 2.10: LOH fragmentation

When a new object is allocated onto the LOH, the Free Space table is checked to see if there is an address range large enough to hold the object. If there is, then an object is allocated at the start byte position and the free space entry is amended.

If there isn't (as in Figure 2.11), then the object will be allocated at the next free space above, which, in this case, is above Object D.

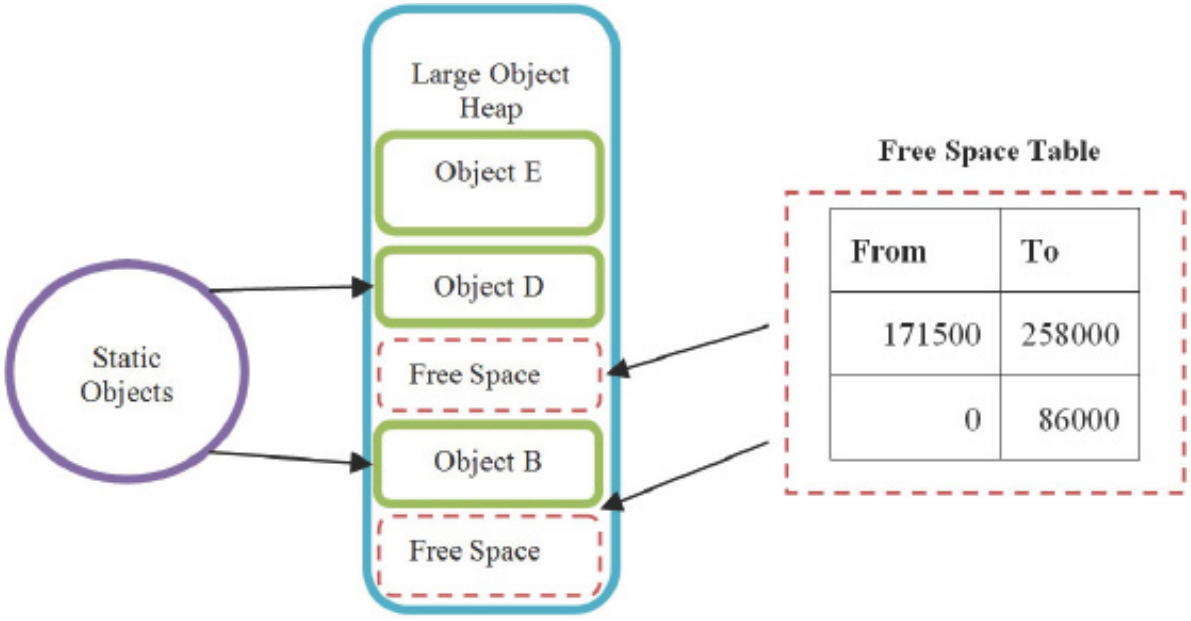

Figure 2.11: LOH allocation fragmentation 1.

It is very unlikely that the objects which will be allocated will be of a size that exactly matches an address range in the Free Space table. As a result, small chunks of memory will almost always be left between objects, resulting in fragmentation.

If the chunks are <85 K, they will be left with no possibility of reuse, as objects of that size obviously never make it onto the LOH. The result is that, as allocation demand increases, new segments are reserved for the LOH, even though space, albeit fragmented space, is still available.

In fact, for performance reasons, .NET preferentially allocates large objects at the end of the heap (i.e. after the last allocated object). When a large object needs to be allocated, .NET has a choice; either run a Gen 2 GC and identify free space on the LOH, or append the object to the end (which may involve extending the heap with additional segments). It tends to go for the second, easier, faster option, and avoids the full GC. This is good for performance, but it is a significant cause of memory fragmentation.

Ultimately, the memory footprint of the application becomes larger than it should be, and eventually out of memory exceptions are thrown.

Server applications are particularly vulnerable to this, as they usually have a high hit rate and, unless they use regular recycling, are

around for a while.

Another reason why they are vulnerable is because, in particular, ASP.NET applications that use `ViewState` can easily generate strings larger than 85 K that go straight onto the LOH.

You know what I said about 85 K?

I can't finish this section on the LOH without giving you the full picture. You know I said that objects >85 KB are allocated onto the LOH? Well, that works for everything except for some internal arrays, such as arrays of type `double` with a size greater than 1,000 (the threshold is different for certain types).

Normally, you would probably expect that an array of doubles would only be allocated onto the LOH when it reached an array size of about 10,600. However, for performance reasons, doubles arrays of size 999 or less allocate onto the SOH, and arrays of 1,000 or above go onto the LOH.

This makes a good quiz question if you're ever hosting a quiz for .NET developers with specialist subjects in the area of .NET memory management, or any Lord of the Rings convention. More importantly, this matters because it affects what objects cause heap fragmentation.