
More on the Heap

Now that we've had our first look at the heap(s), let's dig a little deeper.

When a reference type is created (`class` , `delegate` , `interface` , `string` , or `object`), it's allocated onto the heap. Of the four heaps we've seen so far, .NET uses two of them to manage large objects (anything over 85 K) and small objects differently. They are known as managed heaps.

To make it the worry-free framework that it is, .NET doesn't let you allocate objects directly onto the heap like C/C++ does. Instead, it manages object allocations on your behalf, freeing you from having to de-allocate everything you create. By contrast, if a C++ developer didn't clean up their allocated objects, then the application would just continually leak memory.

To create an object, all you need to do is use the `new` keyword; .NET will take care of creating, initializing and placing the object on the right heap, and reserving any extra memory necessary. After that you can pretty much forget about that object, because you don't have to delete it when you're finished with it.

Naturally, you can help out by setting objects to null when you've finished with them, but most of the time, when an object goes out of scope, it will be cleaned up automatically.

Garbage collection

To achieve this automatic cleanup, .NET uses the famous (or perhaps infamous) **garbage collector** (**GC**). All the GC does is look for allocated objects on the heap that aren't being referenced by anything. The most obvious source of references, as we saw earlier, is the stack. Other potential sources include:

- global/static object references
- CPU registers
- object finalization references (more later)
- Interop references (.NET objects passed to COM/API calls)
- stack references.

Collectively, these are all called root references or GC roots.

As well as root references, an object can also be referenced by other objects. Imagine the classic `Customer` class, which usually has a collection storing `Order` classes.

When an `Order` is added to the order collection, the collection itself then holds a reference to the added order. If the instance of the `Customer` class had a stack reference to it as well, it would have the following references:

- a stack-based root reference for a `Customer` containing:
  - a reference to the orders `ArrayList` collection, which contains:
    - references to `order` objects.

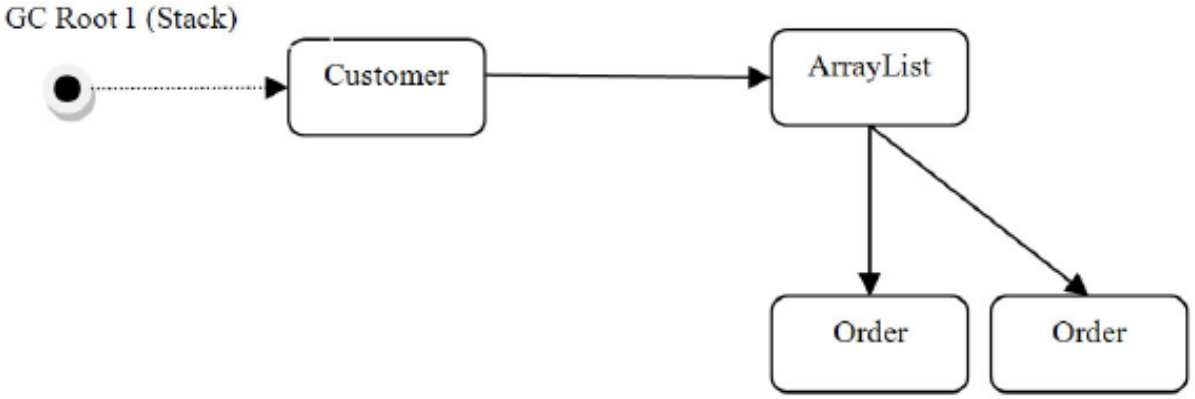

Figure 1.6: Reference tree for a typical scenario.

Figure 1.6 shows the basic reference tree, with a global root reference to a `Customer` class that, in turn, holds a collection of `Order` classes.

This is important because if an object doesn't ultimately have a root reference then it can't actually be accessed by code, so it is no longer in use, and can be removed. As you can see above, a large number of objects can be maintained by just a single root reference, which is both good and bad, as we'll see later.

Inspection and collection

To make sure objects which are no longer in use are cleared away, the GC simply gets a list of all root references and, for each one, moves along its reference tree "marking" each object found as being in use (we'll come back to what that means in just a moment). Any objects not marked as being in use, or "live," are free to be "collected" (which we'll also come back to later).

A simplified version would look something like this:

```
void Collect()
{
   List gcRoots=GetAllGCRoots();
   foreach (objectRef root in gcRoots)
   {
     Mark(root);
   }
   Cleanup();
}
```

Listing 1.11: Simplified GC collection in pseudo code.

The **`Mark`** operation adds an object to an "object still in use" list (if it's not already in there), and then iterates through all of its child object references, marking each one in turn. The result is a list of all objects currently in memory that are still in use (Listing 1.12).

Once that list is compiled, the GC can then go about cleaning up the heaps, and we'll now go through how the **`Cleanup`** operation works differently for both the SOH and LOH. In both cases, the result of a cleanup operation is a resetting of the "object still in use" list, ready for the next collection.

```
Void Mark(objectRef o)
{
   if(!InUseList.Exists(o))
   {
     InUseList.Add(o);
     List refs=GetAllChildReferences(o);
```

```
      foreach (objectRef childRef in refs)
      {
        Mark(childRef);
      }
    }
  }
```

Listing 1.12: Simplified GC `Mark` operation in pseudo code.

SOH cleanup – heap compaction

Garbage collection of the Small Object Heap (SOH) involves compaction. This is because the SOH is a contiguous heap where objects are allocated consecutively on top of each other. When compaction occurs, marked objects are copied over the space taken up by unmarked objects, overwriting those objects, removing any gaps, and keeping the heap contiguous; this process is known as Copy Collection. The advantage of this is that heap fragmentation (i.e. unusable memory gaps) is kept to a minimum. The main disadvantage is that compaction involves copying chunks of memory around, which requires CPU cycles and so, depending on frequency, can cause performance problems. What you gain in efficient allocation you could lose in compaction costs.

LOH sweeping – free space tracking

The Large Object Heap (LOH) isn't compacted, and this is simply because of the time it would take to copy large objects over the top of unused ones. Instead, the LOH keeps track of free and used space, and attempts to allocate new objects into the most appropriately-sized free slots left behind by collected objects.

As a result of this, the LOH is prone to fragmentation, wherein memory gaps are left behind that can only be used if large objects (i.e. >85 KB) of a similar or smaller size to those gaps are subsequently allocated. We will look more closely at these managed heaps in Chapter 2.

It's worth pointing out that the actual algorithms used to carry out garbage collection are known only to the .NET GC team, but one thing I do know is that they will have used every optimization possible.