
What I Didn't Tell You Earlier

You remember the discussion about optimizing garbage collection in the last chapter? Well, I deliberately simplified it (a bit…kind of). Let me explain what I mean.

The problem is that objects aren't always all created at the same time. Often, when objects are created, only some of their member variables are created immediately, with others only instantiating much later. This means that an object can contain references to objects from younger generations, for example, a Gen 2 object referencing a Gen 0 one.

```
class LivesInGen2forAges
{
      private SpannerInWorks _createdLater;
      public void DoABitMoreWork()
     {
              _createdLater=new SpannerInWorks();
     }
}
```

Listing 3.1: Gen 2/Gen 0 reference issue example.

Listing 3.1 shows some code that could create this situation. If `DoABitMoreWork` is called once its object instance has been promoted to Gen 2, then it would immediately hold a Gen 0 reference to the `SpannerInWorks` instance via the `_createdLater` variable.

The reason this is a problem is because of the generational GC optimization I mentioned in Chapter 2, wherein the GC only collects objects residing in the generation it's collecting (and younger). As mentioned earlier, this is faster than a full collection because, as part of the collection process, the GC only needs to inspect the generations currently being collected. As a result, a Gen 0 collection won't consider object references from all of the objects in Gen 1 and Gen 2.

That's a problem because, without looking for all possible references, the GC could assume an object is no longer needed, and allow it to be collected.

Without a work-around in the earlier example, `_createdLater` would be assumed to be rootless and, with only a Gen 2 reference, would be collected in Gen 0.

.NET needs a way to detect when changes are made to object instances in later generations and the good news is that it does have a way to do this.

The card table

A data structure called the card table is used to record when objects are created and referenced from older objects. It's specifically designed to maintain GC performance but still allow objects' references to be tracked regardless of their generation.

When Gen 0 or Gen 1 objects are created from within Gen 2 objects, the execution engine puts an entry in the card table. This check is made before any reference is created, to see if that reference is to a “previous generation” object, and is known as the “write barrier.”

It's tempting to think of the card table structure as having two columns, an object reference id and a bit flag, and having one entry per object. The problem with this picture would be that the huge number of objects that need to be stored would really hit performance.

Instead, .NET stores a bit pattern, with one bit representing 128 bytes of heap memory. If an object residing within the address range of that bit has created a Gen 0 or Gen 1 object, then the bit is set.

After that, when a Gen 0 or Gen 1 collection takes place, any objects residing in the address range of a set bit in the card table are included in the “in use list” inspection, as are all of their child references.

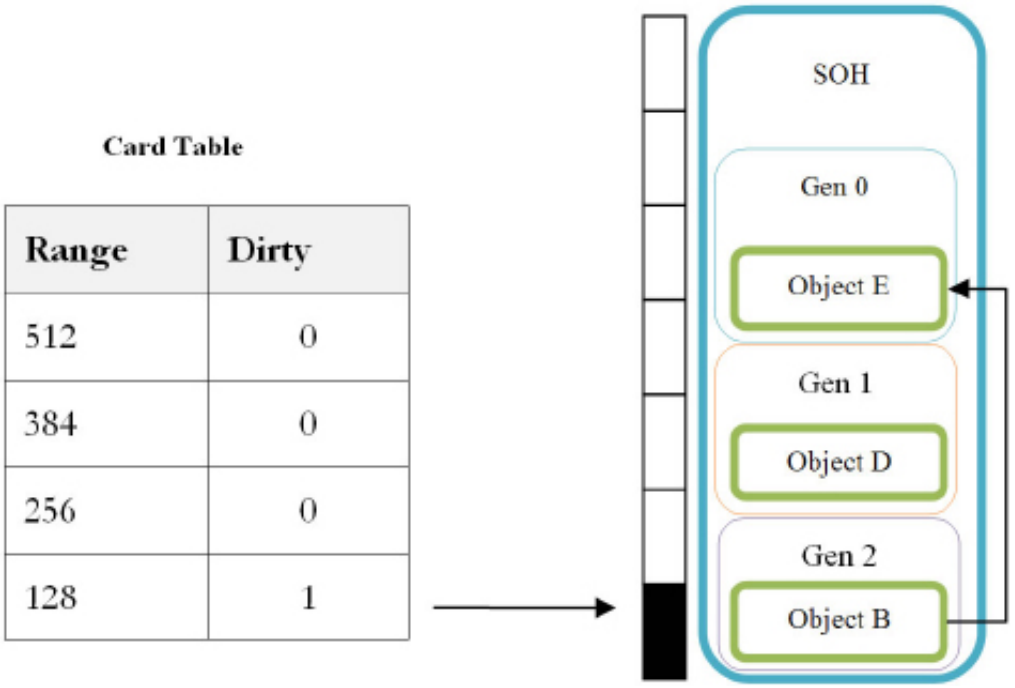

Figure 3.1: Simplified card table.

As we can see in Figure 3.1, Object B instantiates a reference to Object E, and marks the byte portion in the card table in which it itself resides. The next time a Gen 0 collection takes place, all child references for objects located within the pseudo example byte range 0–127 held in the flagged region of the card table will be inspected.