When do I need to use GC.KeepAlive?

devblogs.microsoft.com/oldnewthing/20100813-00

August 13, 2010

Raymond Chen

Finalization is the crazy wildcard in garbage collection. It operates "behind the GC", running after the GC has declared an object dead. Think about it: Finalizers run on objects that have no active references. How can `this` be a reference to an object that has no references? That's just crazy-talk!

Finalizers are a Ouija board, permitting dead objects to operate "from beyond the grave" and affect live objects. As a result, when finalizers are involved, there is a lot of creepy spooky juju going on, and you need to tread very carefully, or your soul will become cursed.

Let's step back and look at a different problem first. Consider this class which doesn't do anything interesting but works well enough for demonstration purposes:

```
class Sample1 {
 private StreamReader sr;
 public Sample1(string file) : sr(new StreamReader(file)) { }
 public void Close() { sr.Close(); }
 public string NextLine() { return sr.ReadLine(); }
}
```

What happens if one thread calls `Sample1.NextLine()` and another thread calls `Sample1.Close()`? If the `NextLine()` call wins the race, then you have a stream closed while it is in the middle of its `ReadLine` method. Probably not good. If the `Close()` call wins the race, then when the `NextLine()` call is made, you end up reading from a closed stream. Definitely not good. Finally, if the `NextLine()` call runs to completion before the `Close()`, then the line is successfully read before the stream is closed.

Having this race condition is clearly an unwanted state of affairs since the result is unpredictable.

Now let's change the `Close()` method to a finalizer.

```
class Sample2 {
 private StreamReader sr;
 public Sample2(string file) : sr(new StreamReader(file)) { }
 ~Sample2() { sr.Close(); }
 public string NextLine() { return sr.ReadLine(); }
}
```

Remember that we learned that an object becomes eligible for garbage collection when there are no active references to it, and that it can happen even while a method on the object is still active. Consider this function:

```
string FirstLine(string fileName) {
 Sample2 s = new Sample2(fileName);
 return s.NextLine();
}
```

We learned that the `Sample2` object becomes eligible for collection during the execution of `NextLine()`. Suppose that the garbage collector runs and collects the object while `NextLine` is still running. This could happen if `ReadLine` takes a long time, say, because the hard drive needs to spin up or there is a network hiccup; or it could happen just because it's not your lucky day and the garbage collector ran at just the wrong moment. Since this object has a finalizer, the finalizer runs before the memory is discarded, and the finalizer closes the `StreamReader`.

Boom, we just hit the race condition we considered when we looked at `Sample1`: The stream was closed while it was being read from. The garbage collector is a rogue thread that closes the stream at a bad time. The problem occurs because the garbage collector doesn't know that the finalizer is going to *make changes to other objects*.

Classically speaking, there are three conditions which in combination lead to this problem:

1. Containment: An entity `a` retains a reference to another entity `b`.
2. Incomplete encapsulation: The entity `b` is visible to an entity outside `a`.
3. Propagation of destructive effect: Some operation performed on entity `a` has an effect on entity `b` which alters its proper usage (usually by rendering it useless).

The first condition (containment) is something you do without a second's thought. If you look at any class, there's a very high chance that it has, among its fields, a reference to another object.

The second condition (incomplete encapsulation) is also a common pattern. In particular, if `b` is an object with methods, it will be visible to itself.

The third condition (propagation of destructive effect) is the tricky one. If an operation on entity `a` has a damaging effect on entity `b`, the code must be careful not to damage it while it's still being used. This is something you usually take care of explicitly, since you're the one

who wrote the code that calls the destructive method.

Unless the destructive method is a finalizer.

If the destructive method is a finalizer, then *you do not have complete control over when it will run*. And it is one of the fundamental laws of the universe that events will occur at the worst possible time.

Enter `GC.KeepAlive()`. The purpose of `GC.KeepAlive()` is to force the garbage collector to treat the object as still live, thereby preventing it from being collected, and thereby preventing the finalizer from running prematurely.

(Here's the money sentence.) You need to use `GC.KeepAlive` when the finalizer for an object has a destructive effect on a contained object.

The problem is that it's not always clear which objects have finalizers which have destructive effect on a contained object. There are some cases where you can suspect this is happening due to the nature of the object itself. For example, if the object manages something external to the CLR, then its finalizer will probably destroy the external object. But there can be other cases where the need for `GC.KeepAlive` is not obvious.

A much cleaner solution than using `GC.KeepAlive` is to use the `IDisposable` interface, formalized by the `using` keyword. Everybody knows that the `using` keyword ensures that the object being used is disposed at the end of the block. But it's also the case (and it is this behavior that is important today) that the `using` keyword also *keeps the object alive until the end of the block*. (Why? Because the object needs to be alive so that we can call `Dispose` on it!)

This is one of the reasons I don't like finalizers. Since they operate underneath the GC, they undermine many principles of garbage collected systems. (See also *resurrection*.) As we saw earlier, a correctly-written program cannot rely on side effects of a finalizer, so in theory all finalizers could be nop'd out without affecting correctness.

The garbage collector purist in me also doesn't like finalizers because they prevent the running time of a garbage collector to be proportional to the amount of *live* data, like say in a classic two-space collector. (There is also a small constant associated with the amount of *dead* data, which means that the overall complexity is proportional to the amount of *total data*.)

If I ruled the world, I would decree that the only thing you can do in a finalizer is perform some tests to ensure that all the associated external resources have already been explicitly released, and if not, raise a fatal exception: `System.Exception.ResourceLeak`.

Bonus reading