Weak References

Weak references have been around since the beginning days of the .NET Framework, but they are still not very well known or understood. In a nutshell, weak references allow you to violate some of the fundamental concepts of the GC. A weak reference allows you to flag to the GC that it is OK to collect an object but still keep access to the data through a `WeakReference` object.

If the only reference to a block of memory is a weak reference, then this memory is eligible for garbage collection. When you need to reference the memory again, you can access the `Target` property of the weak reference. If the `Target` is null, then the object has been collected and will need to be re-created. However, if the `Target` is not null, then you can cast it back to the original type, and you are back to having a strong reference to the memory in question.

Obviously, great care should be taken with weak references. You have to carefully check to ensure that the object wasn't collected and re-create as needed. This is also not a good candidate solution for every memory issue. For example, if your memory issue stems from lots of small objects, you may exacerbate the problem, since the weak reference will require more memory than the original reference. Also, if the data is time-consuming to produce or calculate, you may just create more performance problems. If the data in the object is expensive to produce or time sensitive, a weak reference may make matters worse. Just for clarity, if you have to make an external service call to get the data, it is expensive to produce. If you have to use a slow or unreliable network connection, it is expensive to produce, or if the data depends on geo-tagging details, it may be expensive to produce.

The ideal candidate for a weak reference will be easy to calculate and will take up a substantial amount of memory. Naturally, defining “easy to calculate” and “substantial amount of memory” is very application/hardware specific, so let's rather say that the ideal candidate is something that you only want to have one copy of in your application, but you don't want to explicitly manage its life cycle. To make sure you have some data to work with, whenever you use weak references, you should also add instrumentation/logic to ensure that your weak reference strategy is working. At a minimum, track how often you need to recreate the data and how much time you are spending recreating it.

We can initialize a weak reference with code similar to that in Listing 6.5.

```
private WeakReference _weak;
public void Initialize(string xmlPath)
{
 XmlDocument document = new XmlDocument(); ;
 document.Load(xmlPath);
 _weak = new WeakReference(document);
}
```

Listing 6.5: Initializing a weak reference.

After the `Initialize` method, there are no strong references to the XML document that we loaded; the only reference is the weak reference in the `_weak` variable. If, after we call the `Initialize` method, the GC runs, then the weak reference will be collected and we will have no access to the document. If the GC does not run, then the original document will still be accessible in the `WeakReference` variable.

We can interrogate and use the weak reference with code similar to that shown in Listing 6.6.

```
public void Parse ()
{
  XmlDocument document = null;
  if (_weak.IsAlive )
     document = _weak.Target as XmlDocument;
  if (document == null)
```

```
    {
        // log that weak reference was lost
        // recreate the document
    }
    // handle the parsing
}
```

Listing 6.6: Testing to see if the weak reference is still valid.

If you find that you have to recreate the object frequently, or if your original estimate for how much time it takes to recreate the data is inaccurate, you may want to revisit the choice to use weak references. In almost all cases, you will be better served with reviewing your instrumentation data periodically to ensure that you are still getting the results that you expect. If you don't get the results you expect, you will almost certainly simplify your application logic by removing the weak reference logic.