Weak References

Weak object references allow you to keep hold of objects (as they are another source of GC roots), but still allow them to be collected if the GC needs to. They are a kind of compromise between code performance and memory efficiency, in the sense that creating an object takes CPU time, but keeping it loaded takes memory. In fact, this is particularly relevant for large data structures.

You may have an application allowing the user to browse through large data structures, some of which they may never return to, but some of which they may. So what you could do is convert the strong references to the browsed data structures to weak references, and if the users *do* return to them, that's great. If not, then the GC can reclaim the memory if it needs to.

Listing 3.8 shows a contrived example of using a weak reference: loading a complex data structure, processing it, obtaining a weak reference to it for later reuse, then destroying the strong reference. Let's say that, sometime later, an attempt is made to reclaim the weak reference by attempting to obtain a strong reference from it. However, if `null` is returned from the `Target` property of a weak reference, that means the object has been collected, and so it needs to be recreated.

```
// Load a complex data structure
Complex3dDataStructure floor23=new Complex3dDataStructure();
floor23.Load("floor23.data");

… // Do some work then
// Get a weak reference to it
WeakReference weakRef=new WeakReference(floor23, false);

// Destroy the strong reference, keeping the weak reference
floor23=null;
…
// Some time later try and get a strong reference back
floor23=(Complex3dDataStructure)weakRef.Target;
// recreate if weak ref was reclaimed
if (floor23==null)
{
   floor23=new Complex3dDataStructure();
   floor23.Load("floor23.data");
}
```

Listing 3.8: Weak reference example.

Under the hood

So we've seen how weak references can be used, but now we're going to take a deeper look at them. Weak references are classified as one of:

- short weak references (which ignore finalization)
- long weak references (which consider finalization).

This simply relates to how weak references work with regard to finalizable objects. If you recall from the previous chapter, finalizable objects have an extra pointer on the finalization queue that is used to keep them alive long enough for their finalizer method to be called on the finalizer thread.

Let's see how the two types of weak references differ.

Short weak references

If you create a weak reference to a non-finalizable object or a finalizable object, but pass `false` to the `WeakReference` constructor…

```
WeakReference wr=WeakReference(floor23, false);
```

Listing 3.9.

…then the object reference is added to the **short weak reference table**. These references aren't considered roots, and so, when the GC runs, any references in the short weak reference table that aren't in the GC's "object in use" list are deleted.

Long weak references

On the other hand, if you create a weak reference to a finalizable object and pass `true` to the `WeakReference` constructor…

```
WeakReference wr=WeakReference(floor23, true);
```

Listing 3.10.

…then the object reference is added to the **long weak reference table**. These references also aren't considered roots and, when the GC runs again, any references in the long weak reference table that aren't in either the "object in use" list or the finalization queue can be deleted.