Weak References

Weak references are a supplementary mechanism for handling references to managed objects. The typical object reference (also known as *strong reference*) is very deterministic: as long as you have a reference to the object, the object will stay alive. This is the correctness promise of the garbage collector.

However, in some scenarios, we would like to keep an invisible string attached to an object without interfering with the garbage collector's ability to reclaim that object's memory. If the GC reclaimed the memory, our string becomes unattached and we can detect this. If the GC hasn't touched the object yet, we can pull the string and retrieve a strong reference to the object to use it again.

This facility is useful for various scenarios, of which the following are the most common:

- Providing an external service without keeping the object alive. Services such as timers and events can be provided to objects without keeping them referenced, which can solve many typical memory leaks.
- Automatically managing a cache or pool strategy. A cache can keep weak references to the least recently used objects without preventing them from being collected; a pool can be partitioned into a minimum size which contains strong references and an optional size which contains weak references.
- Keeping a large object alive with the hope that it will not be collected. The application can hold a weak reference to a large object that took a long time to create and initialize. The object might be collected, in which case the application will reinitialize it; otherwise, it can be used the next time it is needed.

Weak references are exposed to application code through the System.WeakReference class, which is a special case of the `System.Runtime.InteropServices`.GCHandle type. A weak reference has the IsAlive Boolean property which indicates whether the underlying object hasn't been collected yet, and the Target property that can be used to retrieve the underlying object (if it has been collected, null is returned).

Caution Note that the only safe way of obtaining a strong reference to the target of a weak reference is by using the Target property. If the IsAlive property returns true, it is possible that immediately afterwards the object will be collected. To defend against this race condition, you must use the Target property, assign the returned value to a strong reference (local variable, field etc.) and then check whether the returned value is null. Use the IsAlive property when you are interested only in the case that the object is dead; for example, to remove the weak reference from a cache.

The following code shows a draft implementation of an event based on weak references (see Figure 4-19). The event itself can't use a .NET delegate directly, because a delegate has a strong reference to its target and this is not customizable. However, it can store the delegate's target (as a weak reference) and its method. This prevents one of the most common .NET memory leaks—forgetting to unregister from an event!

```
public class Button {
  private class WeakDelegate {
    public WeakReference Target;
    public MethodInfo Method;
  }
  private List<WeakDelegate> clickSubscribers = new List<WeakDelegate>();

  public event EventHandler Click {
    add {
     clickSubscribers.Add(new WeakDelegate {
     Target = new WeakReference(value.Target),
     Method = value.Method
     });
    }
    remove {
     //...Implementation omitted for brevity
    }
  }
  public void FireClick() {
    List<WeakDelegate> toRemove = new List<WeakDelegate>();
    foreach (WeakDelegate subscriber in clickSubscribers) {
     object target = subscriber.Target.Target;
```

```
        if (target == null) {
        toRemove.Add(subscriber);
        } else {
        subscriber.Method.Invoke(target, new object[] { this, EventArgs.Empty });
        }
      }
      clickSubscribers.RemoveAll(toRemove);
    }
}
```

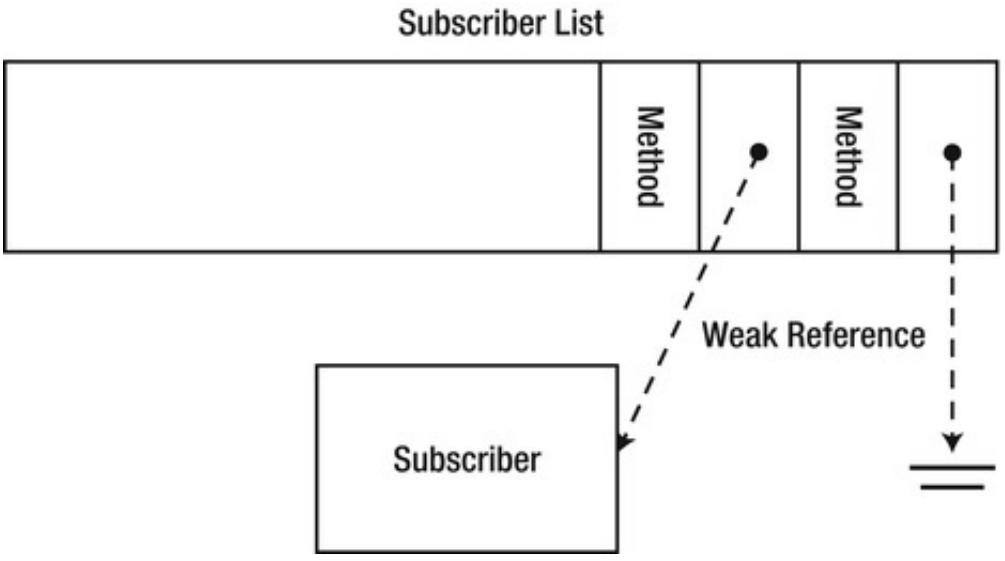

Figure 4-19. The weak event has a weak reference to every subscriber. If the subscriber is unreachable by the application, the weak event can detect this because the weak reference is nulled out

Weak references do not track object resurrection by default. To enable resurrection tracking, use the overloaded constructor that accepts a Boolean parameter and pass true to indicate that resurrection should be tracked. Weak references that track resurrection are called *long weak references*; weak references that do not track resurrection are called *short weak references*.

GC HANDLES

Weak references are a special case of *GC handles*. A GC handle is a special low-level value type that can provide several facilities for referencing objects:

- Keeping a standard (strong) reference to an object, preventing it from being collected. This is represented by the GCHandleType.Normal enumeration value.
- Keeping a short weak reference to an object. This is represented by the GCHandleType.Weak enumeration value.
- Keeping a long weak reference to an object. This is represented by the GCHandleType.WeakTrackResurrection enumeration value.
- Keeping a reference to an object, pinning it so that it cannot move in memory and obtaining its address if necessary. This is represented by the GCHandleType.Pinned enumeration value.

There is rarely any need to use GC handles directly, but they often come up in profiling results as another type of root that can retain managed objects.