Presenting WeakDictionary[TKey, TValue]

[MSDNArchive](http://blogs.msdn.com/274986/ProfileUrlRedirect.ashx)

**4 Jun 2006 8:14 PM**

* [10](http://blogs.msdn.com/b/nicholg/archive/2006/06/04/617466.aspx#comments#comments)

This morning, I posted a [base class](http://blogs.msdn.com/nicholg/archive/2006/06/04/616787.aspx) to assist with implementing IDictionary<TKey,TValue>. As I mentioned this was just the first chunk of work required as part of my goal of building a generic dictionary which uses weak references for its keys and values.

While it's reasonably easy to use WeakReference as the TValue in Dictionary<TKey, TValue> using WeakReference as TKey is actually very tricky to get right. I've aimed to hide the heavy lifting behind the WeakDictionary abstraction so that it can be used as any other IDictionary<TKey,TValue> with the caveat that entries will disappear if either the key or value can be reclaimed by the GC. This means that you don't ever have to work directly with weak references and you can just map keys to values and the dictionary will use weak references internally to ensure that both can still be collected.

The dictionary implementation depends on a few interesting helper classes. Let's look at them first:

// Adds strong typing to WeakReference.Target using generics. Also,  
// the Create factory method is used in place of a constructor  
// to handle the case where target is null, but we want the   
// reference to still appear to be alive.  
internal class WeakReference<T> : WeakReference where T : class {  
    public static WeakReference<T> Create(T target) {  
        if (target == null)  
            return WeakNullReference<T>.Singleton;  
  
        return new WeakReference<T>(target);  
    }  
  
    protected WeakReference(T target)  
        : base(target, false) { }  
  
    public new T Target {  
        get { return (T)base.Target; }  
    }  
}  
  
// Provides a weak reference to a null target object, which, unlike  
// other weak references, is always considered to be alive. This   
// facilitates handling null dictionary values, which are perfectly  
// legal.  
internal class WeakNullReference<T> : WeakReference<T> where T : class {  
    public static readonly WeakNullReference<T> Singleton = newWeakNullReference<T>();  
      
    private WeakNullReference() : base(null) { }  
      
    public override bool IsAlive {  
        get { return true; }  
    }  
}  
  
// Provides a weak reference to an object of the given type to be used in  
// a WeakDictionary along with the given comparer.  
internal sealed class WeakKeyReference<T> : WeakReference<T> where T : class {  
    public readonly int HashCode;  
      
    public WeakKeyReference(T key, WeakKeyComparer<T> comparer)  
        : base(key) {  
        // retain the object's hash code immediately so that even  
        // if the target is GC'ed we will be able to find and  
        // remove the dead weak reference.  
        this.HashCode = comparer.GetHashCode(key);  
    }  
}  
  
// Compares objects of the given type or WeakKeyReferences to them  
// for equality based on the given comparer. Note that we can only  
// implement IEqualityComparer<T> for T = object as there is no   
// other common base between T and WeakKeyReference<T>. We need a  
// single comparer to handle both types because we don't want to  
// allocate a new weak reference for every lookup.  
internal sealed class WeakKeyComparer<T> : IEqualityComparer<object>  
    where T : class {  
  
    private IEqualityComparer<T> comparer;  
  
    internal WeakKeyComparer(IEqualityComparer<T> comparer) {  
        if (comparer == null)  
            comparer = EqualityComparer<T>.Default;  
  
        this.comparer = comparer;  
    }  
  
    public int GetHashCode(object obj) {  
        WeakKeyReference<T> weakKey = obj as WeakKeyReference<T>;  
        if (weakKey != null) return weakKey.HashCode;  
        return this.comparer.GetHashCode((T)obj);  
    }  
  
    // Note: There are actually 9 cases to handle here.  
    //  
    //  Let Wa = Alive Weak Reference  
    //  Let Wd = Dead Weak Reference  
    //  Let S  = Strong Reference  
    //    
    //  x  | y  | Equals(x,y)  
    // -------------------------------------------------  
    //  Wa | Wa | comparer.Equals(x.Target, y.Target)   
    //  Wa | Wd | false  
    //  Wa | S  | comparer.Equals(x.Target, y)  
    //  Wd | Wa | false  
    //  Wd | Wd | x == y  
    //  Wd | S  | false  
    //  S  | Wa | comparer.Equals(x, y.Target)  
    //  S  | Wd | false  
    //  S  | S  | comparer.Equals(x, y)  
    // -------------------------------------------------  
    public new bool Equals(object x, object y) {  
        bool xIsDead, yIsDead;  
        T first  = GetTarget(x, out xIsDead);  
        T second = GetTarget(y, out yIsDead);  
  
        if (xIsDead)  
            return yIsDead ? x == y : false;  
  
        if (yIsDead)  
            return false;  
          
        return this.comparer.Equals(first, second);  
    }  
  
    private static T GetTarget(object obj, out bool isDead) {  
        WeakKeyReference<T> wref = obj as WeakKeyReference<T>;  
        T target;  
        if (wref != null) {  
            target = wref.Target;  
            isDead = !wref.IsAlive;  
        }  
        else {  
            target = (T)obj;  
            isDead = false;  
        }  
        return target;  
    }  
}

Getting the comparer right is actually the hardest problem to solve. Once it's out of the way, the rest is just plumbing. Here's the code for the dictionary itself:

[**Update:**I renamed SetEntry to SetValue for symmetry with TryGetValue.]

/// <summary>  
/// A generic dictionary, which allows both its keys and values   
/// to be garbage collected if there are no other references  
/// to them than from the dictionary itself.  
/// </summary>  
///   
/// <remarks>  
/// If either the key or value of a particular entry in the dictionary  
/// has been collected, then both the key and value become effectively  
/// unreachable. However, left-over WeakReference objects for the key  
/// and value will physically remain in the dictionary until  
/// RemoveCollectedEntries is called. This will lead to a discrepancy  
/// between the Count property and the number of iterations required  
/// to visit all of the elements of the dictionary using its  
/// enumerator or those of the Keys and Values collections. Similarly,  
/// CopyTo will copy fewer than Count elements in this situation.  
/// </remarks>  
public sealed class WeakDictionary<TKey, TValue> : BaseDictionary<TKey, TValue>  
    where TKey : class  
    where TValue : class {  
  
    private Dictionary<object, WeakReference<TValue>> dictionary;  
    private WeakKeyComparer<TKey> comparer;  
  
    public WeakDictionary()  
        : this(0, null) { }  
  
    public WeakDictionary(int capacity)  
        : this(capacity, null) { }  
  
    public WeakDictionary(IEqualityComparer<TKey> comparer)  
        : this(0, comparer) { }  
  
    public WeakDictionary(int capacity, IEqualityComparer<TKey> comparer) {  
        this.comparer = new WeakKeyComparer<TKey>(comparer);  
        this.dictionary = new Dictionary<object, WeakReference<TValue>>(capacity,this.comparer);  
    }  
  
    // WARNING: The count returned here may include entries for which  
    // either the key or value objects have already been garbage  
    // collected. Call RemoveCollectedEntries to weed out collected  
    // entries and update the count accordingly.  
    public override int Count {  
        get { return this.dictionary.Count; }  
    }  
  
    public override void Add(TKey key, TValue value) {

        if (key == null) throw new ArgumentNullException("key");   
        WeakReference<TKey> weakKey = new WeakKeyReference<TKey>(key, this.comparer);  
        WeakReference<TValue> weakValue = WeakReference<TValue>.Create(value);  
        this.dictionary.Add(weakKey, weakValue);  
    }  
  
    public override bool ContainsKey(TKey key) {  
        return this.dictionary.ContainsKey(key);  
    }  
  
    public override bool Remove(TKey key) {  
        return this.dictionary.Remove(key);  
    }  
  
    public override bool TryGetValue(TKey key, out TValue value) {  
        WeakReference<TValue> weakValue;  
        if (this.dictionary.TryGetValue(key, out weakValue)) {  
            value = weakValue.Target;  
            return weakValue.IsAlive;  
        }  
        value = null;  
        return false;  
    }  
  
    protected override void SetValue(TKey key, TValue value) {  
        WeakReference<TKey> weakKey = new WeakKeyReference<TKey>(key, this.comparer);  
        this.dictionary[weakKey] = WeakReference<TValue>.Create(value);  
    }  
  
    public override void Clear() {  
        this.dictionary.Clear();  
    }  
  
    public override IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator() {  
        foreach (KeyValuePair<object, WeakReference<TValue>> kvp in this.dictionary) {  
            WeakReference<TKey> weakKey = (WeakReference<TKey>)(kvp.Key);  
            WeakReference<TValue> weakValue = kvp.Value;  
            TKey key = weakKey.Target;  
            TValue value = weakValue.Target;  
            if (weakKey.IsAlive && weakValue.IsAlive)  
                yield return new KeyValuePair<TKey, TValue>(key, value);  
        }  
    }  
  
    // Removes the left-over weak references for entries in the dictionary  
    // whose key or value has already been reclaimed by the garbage  
    // collector. This will reduce the dictionary's Count by the number  
    // of dead key-value pairs that were eliminated.  
    public void RemoveCollectedEntries() {  
        List<object> toRemove = null;  
        foreach (KeyValuePair<object, WeakReference<TValue>> pair in this.dictionary) {  
            WeakReference<TKey> weakKey = (WeakReference<TKey>)(pair.Key);  
            WeakReference<TValue> weakValue = pair.Value;  
  
            if (!weakKey.IsAlive || !weakValue.IsAlive) {  
                if (toRemove == null)  
                    toRemove = new List<object>();  
                toRemove.Add(weakKey);  
            }  
        }  
  
        if (toRemove != null) {  
            foreach (object key in toRemove)  
                this.dictionary.Remove(key);  
        }  
    }  
}