Implementing IDictionary[TKey, TValue] Isn't Trivial

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



**4 Jun 2006 7:39 AM**

* [3](http://blogs.msdn.com/b/nicholg/archive/2006/06/04/616787.aspx#comments#comments)

I recently needed a dictionary which uses weak references to store both its keys and values, but which otherwise tried its hardest to look and feel just like any other IDictionary<TKey, TValue>. Since there's no such collection available in the BCL, I set out to build  it. I soon discovered that there's a fair bit of plumbing required to implement IDictionary<TKey,TValue> if your underlying storage is at all different from Dictionary<TKey,TValue>. I decided to factor the work out in to two separate chunks. The first chunk was to write an abstract base class which provides as much of the plumbing as possible and defers to the subclass wherever the underlying representation is critical to the implementation.

The class reduces the task of building a new dictionary down to the following abstract methods:

   int Count { get; }  
   void Clear();  
   void Add(TKey key, TValue value);  
   bool ContainsKey(TKey key);  
   bool Remove(TKey key);  
   bool TryGetValue(TKey key, out TValue value);  
   IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator();  
   void SetValue(TKey key, TValue value);

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

Anyhow, here's the code. Let me know what you think....

/// <summary>  
/// Represents a dictionary mapping keys to values.  
/// </summary>  
///   
/// <remarks>  
/// Provides the plumbing for the portions of IDictionary<TKey,  
/// TValue> which can reasonably be implemented without any  
/// dependency on the underlying representation of the dictionary.  
/// </remarks>  
[DebuggerDisplay("Count = {Count}")]  
[DebuggerTypeProxy(PREFIX + "DictionaryDebugView`2" + SUFFIX)]  
public abstract class BaseDictionary<TKey, TValue> : IDictionary<TKey, TValue> {  
    private const string PREFIX = "System.Collections.Generic.Mscorlib\_";  
    private const string SUFFIX =",mscorlib,Version=2.0.0.0,Culture=neutral,PublicKeyToken=b77a5c561934e089";  
  
    private KeyCollection keys;  
    private ValueCollection values;  
  
    protected BaseDictionary() { }  
  
    public abstract int Count { get; }  
    public abstract void Clear();  
    public abstract void Add(TKey key, TValue value);  
    public abstract bool ContainsKey(TKey key);  
    public abstract bool Remove(TKey key);  
    public abstract bool TryGetValue(TKey key, out TValue value);  
    public abstract IEnumerator<KeyValuePair<TKey, TValue>> GetEnumerator();  
    protected abstract void SetValue(TKey key, TValue value);  
  
    public bool IsReadOnly {  
        get { return false; }  
    }  
  
    public ICollection<TKey> Keys {  
        get {  
            if (this.keys == null)  
                this.keys = new KeyCollection(this);  
  
            return this.keys;  
        }  
    }  
  
    public ICollection<TValue> Values {  
        get {  
            if (this.values == null)  
                this.values = new ValueCollection(this);  
  
            return this.values;  
        }  
    }  
  
    public TValue this[TKey key] {  
        get {  
            TValue value;  
            if (!this.TryGetValue(key, out value))  
                throw new KeyNotFoundException();  
  
            return value;  
        }  
        set {  
            SetValue(key, value);  
        }  
    }  
  
    public void Add(KeyValuePair<TKey, TValue> item) {  
        this.Add(item.Key, item.Value);  
    }  
  
    public bool Contains(KeyValuePair<TKey, TValue> item) {  
        TValue value;  
        if (!this.TryGetValue(item.Key, out value))  
            return false;  
  
        return EqualityComparer<TValue>.Default.Equals(value, item.Value);  
    }  
  
    public void CopyTo(KeyValuePair<TKey, TValue>[] array, int arrayIndex) {  
        Copy(this, array, arrayIndex);  
    }  
  
    public bool Remove(KeyValuePair<TKey, TValue> item) {  
        if (!this.Contains(item))  
            return false;

        return this.Remove(item.Key);  
    }  
  
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {  
        return this.GetEnumerator();  
    }  
  
    private abstract class Collection<T> : ICollection<T> {  
        protected readonly IDictionary<TKey, TValue> dictionary;  
  
        protected Collection(IDictionary<TKey, TValue> dictionary) {  
            this.dictionary = dictionary;  
        }  
  
        public int Count {  
            get { return this.dictionary.Count; }  
        }  
  
        public bool IsReadOnly {  
            get { return true; }  
        }  
  
        public void CopyTo(T[] array, int arrayIndex) {  
            Copy(this, array, arrayIndex);  
        }  
  
        public virtual bool Contains(T item) {  
            foreach (T element in this)  
                if (EqualityComparer<T>.Default.Equals(element, item))  
                    return true;  
            return false;  
        }  
  
        public IEnumerator<T> GetEnumerator() {  
            foreach (KeyValuePair<TKey, TValue> pair in this.dictionary)  
                yield return GetItem(pair);  
        }  
  
        protected abstract T GetItem(KeyValuePair<TKey, TValue> pair);  
  
        public bool Remove(T item) {  
            throw new NotSupportedException("Collection is read-only.");  
        }  
  
        public void Add(T item) {  
            throw new NotSupportedException("Collection is read-only.");  
        }  
  
        public void Clear() {  
            throw new NotSupportedException("Collection is read-only.");  
        }  
  
        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() {  
            return this.GetEnumerator();  
        }  
    }  
  
    [DebuggerDisplay("Count = {Count}")]  
    [DebuggerTypeProxy(PREFIX + "DictionaryKeyCollectionDebugView`2" + SUFFIX)]  
    private class KeyCollection : Collection<TKey> {  
        public KeyCollection(IDictionary<TKey, TValue> dictionary)  
            : base(dictionary) { }  
  
        protected override TKey GetItem(KeyValuePair<TKey, TValue> pair) {  
            return pair.Key;  
        }  
        public override bool Contains(TKey item) {  
            return this.dictionary.ContainsKey(item);  
        }  
    }  
  
    [DebuggerDisplay("Count = {Count}")]  
    [DebuggerTypeProxy(PREFIX + "DictionaryValueCollectionDebugView`2" + SUFFIX)]  
    private class ValueCollection : Collection<TValue> {  
        public ValueCollection(IDictionary<TKey, TValue> dictionary)  
            : base(dictionary) { }  
  
        protected override TValue GetItem(KeyValuePair<TKey, TValue> pair) {  
            return pair.Value;  
        }  
    }  
  
    private static void Copy<T>(ICollection<T> source, T[] array, int arrayIndex) {  
        if (array == null)  
            throw new ArgumentNullException("array");  
  
        if (arrayIndex < 0 || arrayIndex > array.Length)  
            throw new ArgumentOutOfRangeException("arrayIndex");  
  
        if ((array.Length - arrayIndex) < source.Count)  
            throw new ArgumentException("Destination array is not large enough. Check array.Length and arrayIndex.");  
  
        foreach (T item in source)  
            array[arrayIndex++] = item;  
    }  
}