Power Collections Specification

DRAFT COPY

1 Introduction

The Power Collections Library is an open source collection class library for the next version of .NET. The philosophy of the library is to extend the base class library (BCL) of collection classes that will be available with .NET 2.0 rather than provide a complete and separate set of classes from .NET.

The success of the Power Collections project depends on the aid of the community. It is the intentions of the project to have the community aid in the concept as well as its growth. Furthermore, it is intended that the initial version is intended only as a seed that is to be germinated by the community.

As a result, we expect developers to review this document and submit change requests to the Power Collection Class Discussion Forum (located at http://www.wintellect.com/forum/main.asp?CAT_ID=7). These suggestions will be digested on a regular basis and any impacting changes will be reviewed and accepted into the specification and a new version of the library. The best efforts will be made to attribute credit for suggestions.

2 Collection Classes

2.1 Set<ElementType>

A hashed collection of objects, without duplicates, and without an ordering. Adding, removing, and testing membership is fast (near constant time). Similar to a HashTable with no value type, but additionally having set operations like Union, Intersection, Difference.

2.2 Bag<ElementType>

A hashed collection of objects, allowing duplicates, and without an ordering. Adding, removing, and testing membership is fast (near constant time).

2.3 MultiDictionary<KeyType, ValueType>

A hashed collection of key-value pairs, allowing duplicate keys. No ordering is maintains. Adding, removing, testing members, and obtaining all values for a key is fast (near constant time).

2.4 OrderedDictionary<KeyType, ValueType>

A set of key-value pairs, where the keys having an ordering (via IComparable, IComparer, or a supplied comparison function). Implemented with a balanced binary tree (red-black tree). No duplicate keys. Adding, removing, testing membership is O(log n). Can efficiently enumerate keys in order, or enumerate a sub-range (all keys between x and y).

2.5 OrderedMultiDictionary<KeyType, ValueType>

A set of key-value pairs, where the keys having an ordering (via IComparable, IComparer, or a supplied comparison function). Implemented with a balanced binary tree (red-black tree). Duplicate keys are allowed. Adding, removing, testing membership is O(log n). Can efficiently enumerate keys in order, or enumerate a sub-range (all keys between x and y).

2.6 OrderedSet<ElementType>

A set of objects, where the objects have an ordering (via IComparable, IComparer, or a supplied comparison function). Implemented with a balanced binary tree (red-black tree). Duplicate keys are not allowed. Adding, removing, testing membership is O(log n). Can efficiently enumerate keys in order, or enumerate a sub-range (all keys between x and y).

2.7 OrderedBag<ElementType>

A set of objects, where the objects have an ordering (via IComparable, IComparer, or a supplied comparison function). Implemented with a balanced binary tree (red-black tree). Duplicate keys are allowed. Adding, removing, testing membership is O(log n). Can efficiently enumerate keys in order, or enumerate a sub-range (all keys between x and y).

2.8 Summary of the above collections:

Name	Store values with keys?	Allow duplicates?	Use hash function or comparing function?
`Dictionary`	Yes	No	Hash
`MultiDictionary`	Yes	Yes	Hash
`Set`	No	No	Hash
`Bag`	No	Yes	Hash
`OrderedDictionary`	Yes	No	Comparing
`OrderedMultiDictionary`	Yes	Yes	Comparing
`OrderedSet`	No	No	Comparing
`OrderedBag`	No	Yes	Comparing

2.9 Deque

Double-ended queue. Like a queue and stack combined. Allows adding or removing at both ends in fast constant time. Also allows indexing and implements IList. Implemented with an array managed as a ring buffer. The array is grown or shrinks as necessary.

2.10 PriorityQueue

A collection that allows adding objects with an associated priority, as well as removing the item with the highest priority (both are O(log n)) Implemented with a heap structure in an array that grows as needed. Although an OrderedSet can also be efficiently used a priority queue with O(log n) adding and removing, this collection provides a more efficient implementation, better memory usage, at the expense of the generality that OrderedSet provides.

3 Utility Structures

The following utility structs are provided for managing simple pairs and triples of values. For example, a dictionary with two keys can be created be simply making

Dictionary<Pair<string,int>, object>

or an ordered set with

OrderedSet<ComparablePair<string,int>>

3.1 Pair<FirstType, SecondType>

Provides a simple mechanism for storing a pair of values. Equals, GetHashCode, and ToString "do the right thing" using the corresponding functions on the elements.

3.2 Triple<FirstType, SecondType, ThirdType>

Provides a simple mechanism for storing a triple of values. Equals, GetHashCode, and ToString "do the right thing" using the corresponding functions on the elements.

4 Static Methods

The following algorithms would be provided as static methods. They would work on collections through the collection interfaces, primarily IEnumerable and IList.

4.1 IEnumerable<U> Map(IEnumerable<T> collection, Mapping<T, U> func)

Maps all the elements of a collection via a mapping function.

4.2 IEnumerable<T> ReplaceCopy(IEnumerable<T> collection, T value1, T value2)

Replaces all instances of "value1" with "value2".

4.3 IEnumerable<T> ReplaceCopyIf(IEnumerable<T> collection, Predicate<T> func, T value)

Replaces all instances where func returns true with "value".

4.4 IEnumerable<T> RemoveCopy(IEnumerable<T> collection, T value)

Removes all elements equal to “value”.

4.5 IEnumerable<T> RemoveCopyIf(IEnumerable<T> collection, Predicate<T> func)

Removes all elements where func returns true with “value”.

4.6 IEnumerable<T> Filter(IEnumerable<T> collection, Predicate<T> func)

Filters all the elements of a collection, returning all that match a predicate.

4.7 T FindFirst(IEnumerable<T> collection, Predicate<T> func)

Finds the first element of a collection that matches a predicate.

4.8 bool Any(IEnumerable<T> collection, Predicate<T> func)

Returns true if any of the elements of the collection match the predicate.

4.9 bool All(IEnumerable<T> collection, Predicate<T> func)

Returns true if all of the elements of the collection match the predicate.

4.10 bool AreEqual(IEnumerable<T> coll1, IEnumerable<T> coll2, optional Equality<T> func)

Returns true if the two collections have the same elements in the same order. Optionally use an equality function.

4.11 IEnumerable Unique(IEnumerable<T> collection, optional Equality<T> equality)

Removes consecutive equal elements from a collection.

4.12 IEnumerable<T> Concatinate(params IEnumerable<T>[] collections)

Concatinates all of the given collections together.

4.13 IEnumerable<T> MergeSorted(params IEnumerable<T>[] collections, optional Comparer<T>)

Merges all of the given sorted collections together into a sorted collection.

4.14 void Copy(IEnumerable<T> coll, IList<T> list, int start)

Copies a collection into a list at a given point.

4.15 void Insert(IEnumerable<T> coll, IList<T> list, int start)

Inserts a collection into a list at a given point.

4.16 void Replace(IList<T> collection, T value1, T value2)

Replaces all instances of “value1” with “value2”.

4.17 void ReplaceIf(IEnumerable<T> collection, Predicate<T> func, T value)

Replaces all instances where func returns true with “value”.

4.18 void Remove(IList<T> collection, T value)

Removes all instances of “value1”.

4.19 void RemoveIf(IEnumerable<T> collection, Predicate<T> func)

Removes all instances where func returns true.

4.20 IList<T> SubList(IList<T> list, int start, int length)

Creates a wrapper list for a sub-part of a list.

4.21 void Rotate(IList<T> list, int amount)

Rotates a list by the given amount (can be negative).

4.22 void Sort(IList<T> list, optional Comparer<T>)

Sorts a list in-place.

4.23 void StableSort(IList<T> list, optional Comparer<T>)

Sorts a list in-place, using a stable algorithm (equal elements stay in their original order).

4.24 int BinarySearch(IList<T> list, T value, optional Comparer<T>)

Performs a binary search on list for value.

4.25 T Smallest(IComparable<T> list, int n, optional Comparer<T>)

Finds the smallest item of a list.

4.26 T Largest(IComparable<T> list, int n, optional Comparer<T>)

Finds the Nth largest item of a list.

4.27 T NthLargest(IList<T> list, int n, optional Comparer<T>)

Finds the Nth largest item of a list.

4.28 void Reverse(IList<T> list)

Reverse a list.

4.29 IList<T> Reversed(IList<T> list)

Create a reversed wrapper for a list.

4.30 IEnumerable<IList<T>> AllPermutations(IEnumerable<T>)

Create all permutations of the given collection.

4.31 void Shuffle(IList<T>, Random randgenerator)

Shuffle the given list randomly.