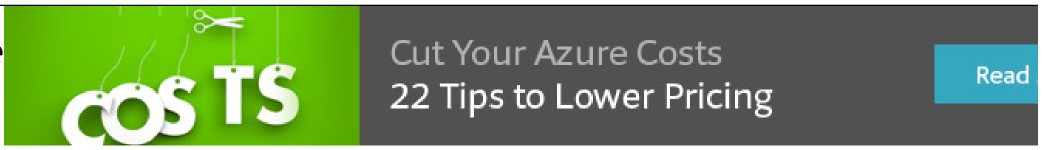
James Michael Hare

...hare-brained ideas from the realm of software development...



<< Automating C# Coding Standards Using FxCop and StyleCop | Home | A Trio of Presentations: Little Wonders, StyleCop, and LINQ/Lambdas

C#/.NET Fundamentals: Choosing the Right Collection Class

The .NET Base Class Library (BCL) has a wide array of collection classes at your disposal which make it easy to manage collection objects. While it's great to have so many classes available, it can be daunting to choose the right collection to use for any given situation. As hard as it may be, choosing the right collection can be absolutely key to the performance and maintainability of your application!

This post will look at breaking down any confusion between each collection and the situations in which they excel. We will be spending most of our time looking at the **System.Collections.Generic** namespace, which is the recommended set of collections.

The Generic Collections: System.Collections.Generic namespace

The generic collections were introduced in .NET 2.0 in the System.Collections.Generic namespace. This is the main body of collections you should tend to focus on first, as they will tend to suit 99% of your needs right up front.

It is important to note that the generic collections are unsynchronized. This decision was made for performance reasons because depending on how you are using the collections its completely possible that synchronization may not be required or may be needed on a higher level than simple method-level synchronization. Furthermore, concurrent read access (all writes done at beginning and never again) is always safe, but for concurrent mixed access you should either synchronize the collection or use one of the concurrent collections.

So let's look at each of the collections in turn and its various pros and cons, at the end we'll summarize with a table to help make easier to compare and contrast the different collections.

The Associative Collection Classes

Associative collections store a value in the collection by providing a key that is used to add/remove/lookup the item. Hence, the container associates the value with the key. These collections are most useful when you need to lookup/manipulate a collection usi key value. For example, if you wanted to look up an order in a collection of orders by an order id, you might have an associative collection where they key is the order id and the value is the order.

The **Dictionary<TKey,TVale>** is probably the most used associative container class. The **Dictionary<TKey,TValue>** is the fastes class for associative lookups/inserts/deletes because it uses a hash table under the covers. Because the keys are hashed, the key should correctly implement **GetHashCode()** and **Equals()** appropriately or you should provide an external **IEqualityComparer** to dictionary on construction. The insert/delete/lookup time of items in the dictionary is amortized constant time - O(1) - which mean matter how big the dictionary gets, the time it takes to find something remains relatively constant. This is highly desirable for high speed lookups. The only downside is that the dictionary, by nature of using a hash table, is unordered, so you cannot easily travers items in a Dictionary in order.

The **SortedDictionary<TKey,TValue>** is similar to the **Dictionary<TKey,TValue>** in usage but very different in implementation. **SortedDictionary<TKey,TValye>** uses a binary tree under the covers to maintain the items in order by the key. As a consequenc sorting, the type used for the key must correctly implement **IComparable<TKey>** so that the keys can be correctly sorted. The s dictionary trades a little bit of lookup time for the ability to maintain the items in order, thus insert/delete/lookup times in a sorted dictionary are logarithmic - O(log n). Generally speaking, with logarithmic time, you can double the size of the collection and it onl to perform one extra comparison to find the item. Use the **SortedDictionary<TKey,TValue>** when you want fast lookups but also to be able to maintain the collection in order by the key.

The **SortedList<TKey,TValue>** is the other sorted associative container class in the generic containers. Once again **SortedList<TKey,TValue>**, like **SortedDictionary<TKey,TValue>**, uses a key to sort key-value pairs. Unlike **SortedDictionary** however, items in a **SortedList** are stored as sorted array of items. This means that insertions and deletions are linear - O(n) - be deleting or adding an item may involve shifting all items up or down in the list. Lookup time, however is O(log n) because the **SortedList** can use a binary search to find any item in the list by its key. So why would you ever want to do this? Well, the answe that if you are going to load the **SortedList** up-front, the insertions will be slower, but because array indexing is faster than follow object links, lookups are marginally faster than a **SortedDictionary**. Once again I'd use this in situations where you want fast look and want to maintain the collection in order by the key, and where insertions and deletions are rare.

The Non-Associative Containers

The other container classes are non-associative. They don't use keys to manipulate the collection but rely on the object itself being stored or some other means (such as index) to manipulate the collection.

The **List<T>** is a basic contiguous storage container. Some people may call this a vector or dynamic array. Essentially it is an arra items that grow once its current capacity is exceeded. Because the items are stored contiguously as an array, you can access item the **List<T>** by index very quickly. However inserting and removing in the beginning or middle of the **List<T>** are very costly bec you must shift all the items up or down as you delete or insert respectively. However, adding and removing at the end of a **List<T** an amortized constant operation - O(1). Typically **List<T>** is the standard go-to collection when you don't have any other constrair and typically we favor a **List<T>** even over arrays unless we are sure the size will remain absolutely fixed.

The **LinkedList<T>** is a basic implementation of a doubly-linked list. This means that you can add or remove items in the middle o linked list very quickly (because there's no items to move up or down in contiguous memory), but you also lose the ability to index items by position quickly. Most of the time we tend to favor **List<T>** over **LinkedList<T>** unless you are doing a lot of adding and removing from the collection, in which case a **LinkedList<T>** may make more sense.

The **HashSet<T>** is an unordered collection of unique items. This means that the collection cannot have duplicates and no order is maintained. Logically, this is very similar to having a **Dictionary<TKey,TValue>** where the **TKey** and **TValue** both refer to the sa object. This collection is very useful for maintaining a collection of items you wish to check membership against. For example, if yo receive an order for a given vendor code, you may want to check to make sure the vendor code belongs to the set of vendor codes handle. In these cases a **HashSet<T>** is useful for super-quick lookups where order is not important. Once again, like in Dictionar type T should have a valid implementation of **GetHashCode()** and **Equals()**, or you should provide an appropriate **IEqualityComparer<T>** to the **HashSet<T>** on construction.

The **SortedSet<T>** is to **HashSet<T>** what the **SortedDictionary<TKey,TValue>** is to **Dictionary<TKey,TValue>**. That is, the **SortedSet<T>** is a binary tree where the key and value are the same object. This once again means that adding/removing/lookup logarithmic - O(log n) - but you gain the ability to iterate over the items in order. For this collection to be effective, type T must implement **IComparable<T>** or you need to supply an external **IComparer<T>**.

Finally, the **Stack<T>** and **Queue<T>** are two very specific collections that allow you to handle a sequential collection of objects i very specific ways. The **Stack<T>** is a last-in-first-out (LIFO) container where items are added and removed from the top of the s Typically this is useful in situations where you want to stack actions and then be able to undo those actions in reverse order as nee The **Queue<T>** on the other hand is a first-in-first-out container which adds items at the end of the queue and removes items fron front. This is useful for situations where you need to process items in the order in which they came, such as a print spooler or wai lines.

So that's the basic collections. Let's summarize what we've learned in a quick reference table.

Collection	Ordering	Contiguous Storage?	Direct Access?	Lookup Efficiency	Manipulate Efficiency	Notes
Dictionary	Unordered	Yes	Via Key	Key: O(1)	O(1)	Best for high performance lookups.
SortedDictionary	Sorted	No	Via Key	Key: O(log n)	O(log n)	Compromise of Dictionary speed and ordering, uses binary search tree.
SortedList	Sorted	Yes	Via Key	Key: O(log n)	O(n)	Very similar to SortedDictionary, except tree is implemented in an array, so has faster lookup on preloaded data, but slower loads.
List	User has precise control over element ordering	Yes	Via Index	Index: O(1) Value: O(n)	O(n)	Best for smaller lists where direct access required and no sorting.
LinkedList	User has precise control over element ordering	No	No	Value: O(n)	O(1)	Best for lists where inserting/deleting in middle is common and no direct access required.
HashSet	Unordered	Yes	Via Key	Key: O(1)	O(1)	Unique unordered collection, like a Dictionary except key and value are same object.
SortedSet	Sorted	No	Via Key	Key: O(log n)	O(log n)	Unique sorted collection, like SortedDictionary except key and value are same object.
Stack	LIFO	Yes	Only Top	Top: O(1)	O(1)*	Essentially same as List<T> except only process as LIFO
Queue	FIFO	Yes	Only Front	Front: O(1)	O(1)	Essentially same as List<T> except only process as FIFO

The Original Collections: System.Collections namespace

The original collection classes are largely considered deprecated by developers and by Microsoft itself. In fact they indicate that fo most part you should always favor the generic or concurrent collections, and only use the original collections when you are dealing legacy .NET code.

Because these collections are out of vogue, let's just briefly mention the original collection and their generic equivalents:

- **ArrayList**
  - A dynamic, contiguous collection of objects.
  - Favor the generic collection **List<T>** instead.
- **Hashtable**
  - Associative, unordered collection of key-value pairs of objects.
  - Favor the generic collection **Dictionary<TKey,TValue>** instead.
- **Queue**
  - First-in-first-out (FIFO) collection of objects.
  - Favor the generic collection **Queue<T>** instead.
- **SortedList**
  - Associative, ordered collection of key-value pairs of objects.
  - Favor the generic collection **SortedList<T>** instead.
- **Stack**
  - Last-in-first-out (LIFO) collection of objects.
  - Favor the generic collection **Stack<T>** instead.

In general, the older collections are non-type-safe and in some cases less performant than their generic counterparts. Once again, only reason you should fall back on these older collections is for backward compatibility with legacy code and libraries only.

The Concurrent Collections: System.Collections.Concurrent namespace

The concurrent collections are new as of .NET 4.0 and are included in the System.Collections.Concurrent namespace. These collecti are optimized for use in situations where multi-threaded read and write access of a collection is desired.

The concurrent queue, stack, and dictionary work much as you'd expect. The bag and blocking collection are more unique. Below is summary of each with a link to a blog post I did on each of them.

- **ConcurrentQueue**
  - Thread-safe version of a queue (FIFO).
  - For more information see: C#/.NET Little Wonders: The ConcurrentStack and ConcurrentQueue
- **ConcurrentStack**
  - Thread-safe version of a stack (LIFO).
  - For more information see: C#/.NET Little Wonders: The ConcurrentStack and ConcurrentQueue
- **ConcurrentBag**
  - Thread-safe unordered collection of objects.
  - Optimized for situations where a thread may be bother reader and writer.
  - For more information see: C#/.NET Little Wonders: The ConcurrentBag and BlockingCollection
- **ConcurrentDictionary**
  - Thread-safe version of a dictionary.
  - Optimized for multiple readers (allows multiple readers under same lock).
  - For more information see C#/.NET Little Wonders: The ConcurrentDictionary
- **BlockingCollection**
  - Wrapper collection that implement producers & consumers paradigm.
  - Readers can block until items are available to read.
  - Writers can block until space is available to write (if bounded).
  - For more information see C#/.NET Little Wonders: The ConcurrentBag and BlockingCollection

Summary

The .NET BCL has lots of collections built in to help you store and manipulate collections of data. Understanding how these collectic work and knowing in which situations each container is best is one of the key skills necessary to build more performant code. Choo the wrong collection for the job can make your code much slower or even harder to maintain if you choose one that doesn't perfor well or otherwise doesn't exactly fit the situation.

Remember to avoid the original collections and stick with the generic collections. If you need concurrent access, you can use the generic collections if the data is read-only, or consider the concurrent collections for mixed-access if you are running on .NET 4.0 c higher.

Tweet Technorati Tags:
C#,.NET,Collecitons,Generic,Concurrent,Dictionary,List,Stack,Queue,SortedList,SortedDictionary,HashSet,SortedSet

Share This Post: Short Url: http://wblo.gs/bx5

Print | posted on Thursday, June 16, 2011 7:11 PM | Filed Under [My Blog C# Software .NET Fundamentals]