Collections and Data Structures

Article • 08/12/2022 • 6 minutes to read

Similar data can often be handled more efficiently when stored and manipulated as a collection. You can use the System.Array class or the classes in the System.Collections, System.Collections.Generic, System.Collections.Concurrent, and System.Collections.Immutable namespaces to add, remove, and modify either individual elements or a range of elements in a collection.

There are two main types of collections; generic collections and non-generic collections. Generic collections are type-safe at compile time. Because of this, generic collections typically offer better performance. Generic collections accept a type parameter when they're constructed. They don't require that you cast to and from the Object type when you add or remove items from the collection. In addition, most generic collections are supported in Windows Store apps. Non-generic collections store items as Object, require casting, and most aren't supported for Windows Store app development. However, you might see non-generic collections in older code.

In .NET Framework 4 and later versions, the collections in the System.Collections.Concurrent namespace provide efficient thread-safe operations for accessing collection items from multiple threads. The immutable collection classes in the System.Collections.Immutable namespace (NuGet package) are inherently thread-safe because operations are performed on a copy of the original collection, and the original collection can't be modified.

Common collection features

All collections provide methods for adding, removing, or finding items in the collection. In addition, all collections that directly or indirectly implement the ICollection interface or the ICollection<T> interface share these features:

- **The ability to enumerate the collection**

  .NET collections either implement System.Collections.IEnumerable or System.Collections.Generic.IEnumerable<T> to enable the collection to be iterated through. An enumerator can be thought of as a movable pointer to any element in the collection. The foreach, in statement and the For Each...Next Statement use the enumerator exposed by the GetEnumerator method and hide the complexity of manipulating the enumerator. In addition, any collection that implements System.Collections.Generic.IEnumerable<T> is considered a *queryable type* and can be queried with LINQ. LINQ queries provide a common pattern for accessing data. They're typically more concise and readable than standard `foreach` loops and provide filtering, ordering, and grouping capabilities. LINQ queries can also improve performance. For more information, see LINQ to Objects (C#), LINQ to Objects (Visual Basic),

Parallel LINQ (PLINQ), Introduction to LINQ Queries (C#), and Basic Query Operations (Visual Basic).

- **The ability to copy the collection contents to an array**

  All collections can be copied to an array using the `CopyTo` method. However, the order of the elements in the new array is based on the sequence in which the enumerator returns them. The resulting array is always one-dimensional with a lower bound of zero.

In addition, many collection classes contain the following features:

- **Capacity and Count properties**

  The capacity of a collection is the number of elements it can contain. The count of a collection is the number of elements it actually contains. Some collections hide the capacity or the count or both.

  Most collections automatically expand in capacity when the current capacity is reached. The memory is reallocated, and the elements are copied from the old collection to the new one. This design reduces the code required to use the collection. However, the performance of the collection might be negatively affected. For example, for List<T>, if Count is less than Capacity, adding an item is an O(1) operation. If the capacity needs to be increased to accommodate the new element, adding an item becomes an O(`n`) operation, where `n` is Count. The best way to avoid poor performance caused by multiple reallocations is to set the initial capacity to be the estimated size of the collection.

  A BitArray is a special case; its capacity is the same as its length, which is the same as its count.

- **A consistent lower bound**

  The lower bound of a collection is the index of its first element. All indexed collections in the System.Collections namespaces have a lower bound of zero, meaning they're 0-indexed. Array has a lower bound of zero by default, but a different lower bound can be defined when creating an instance of the **Array** class using Array.CreateInstance.

- **Synchronization for access from multiple threads** (System.Collections classes only).

  Non-generic collection types in the System.Collections namespace provide some thread safety with synchronization; typically exposed through the SyncRoot and IsSynchronized members. These collections aren't thread-safe by default. If you require scalable and efficient multi-threaded access to a collection, use one of the classes in the System.Collections.Concurrent namespace or consider using an immutable collection. For more information, see Thread-Safe Collections.

Choose a collection

In general, you should use generic collections. The following table describes some common collection scenarios and the collection classes you can use for those scenarios. If you're new to generic collections, the following table will help you choose the generic collection that works best for your task:

I want to...	Generic collection options	Non-generic collection options	Thread-safe or immutable collection options
Store items as key/value pairs for quick look-up by key	Dictionary<TKey,TValue>	Hashtable (A collection of key/value pairs that are organized based on the hash code of the key.)	ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
Access items by index	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
Use items first-in-first-out (FIFO)	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>
Use data Last-In-First-Out (LIFO)	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
Access items sequentially	LinkedList<T>	No recommendation	No recommendation
Receive notifications when items are removed or added to the collection. (implements INotifyPropertyChanged and INotifyCollectionChanged)	ObservableCollection<T>	No recommendation	No recommendation
A sorted collection	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TValue> ImmutableSortedSet<T>
A set for mathematical functions	HashSet<T> SortedSet<T>	No recommendation	ImmutableHashSet<T> ImmutableSortedSet<T>

Algorithmic complexity of collections

When choosing a collection class, it's worth considering potential tradeoffs in performance. Use the following table to reference how various mutable collection types compare in algorithmic complexity to their corresponding immutable counterparts. Often immutable collection types are less performant but provide immutability - which is often a valid comparative benefit.

Mutable	Amortized	Worst Case	Immutable	Complexity
`Stack<T>.Push`	O(1)	O(`n`)	`ImmutableStack<T>.Push`	O(1)
`Queue<T>.Enqueue`	O(1)	O(`n`)	`ImmutableQueue<T>.Enqueue`	O(1)
`List<T>.Add`	O(1)	O(`n`)	`ImmutableList<T>.Add`	O(log `n`)
`List<T>.Item[Int32]`	O(1)	O(1)	`ImmutableList<T>.Item[Int32]`	O(log `n`)
`List<T>.Enumerator`	O(`n`)	O(`n`)	`ImmutableList<T>.Enumerator`	O(`n`)
`HashSet<T>.Add`, lookup	O(1)	O(`n`)	`ImmutableHashSet<T>.Add`	O(log `n`)
`SortedSet<T>.Add`	O(log `n`)	O(`n`)	`ImmutableSortedSet<T>.Add`	O(log `n`)
`Dictionary<T>.Add`	O(1)	O(`n`)	`ImmutableDictionary<T>.Add`	O(log `n`)
`Dictionary<T>` lookup	O(1)	O(1) – or strictly O(`n`)	`ImmutableDictionary<T>` lookup	O(log `n`)
`SortedDictionary<T>.Add`	O(log `n`)	O(`n` log `n`)	`ImmutableSortedDictionary<T>.Add`	O(log `n`)

A `List<T>` can be efficiently enumerated using either a `for` loop or a `foreach` loop. An `ImmutableList<T>`, however, does a poor job inside a `for` loop, due to the O(log `n`) time for its indexer. Enumerating an `ImmutableList<T>` using a `foreach` loop is efficient because `ImmutableList<T>` uses a binary tree to store its data instead of an array like `List<T>` uses. An array can be quickly indexed into, whereas a binary tree must be walked down until the node with the desired index is found.

Additionally, `SortedSet<T>` has the same complexity as `ImmutableSortedSet<T>` because they both use binary trees. The significant difference is that `ImmutableSortedSet<T>` uses an immutable binary tree. Since `ImmutableSortedSet<T>` also offers a System.Collections.Immutable.ImmutableSortedSet<T>.Builder class that allows mutation, you can have both immutability and performance.

Related articles

Title	Description
Selecting a Collection Class	Describes the different collections and helps you select one for your scenario.
Commonly Used Collection Types	Describes commonly used generic and non-generic collection types such as System.Array, System.Collections.Generic.List<T>, and System.Collections.Generic.Dictionary<TKey,TValue>.
When to Use Generic Collections	Discusses the use of generic collection types.
Comparisons and Sorts Within Collections	Discusses the use of equality comparisons and sorting comparisons in collections.
Sorted Collection Types	Describes sorted collections performance and characteristics.
Hashtable and Dictionary Collection Types	Describes the features of generic and non-generic hash-based dictionary types.
Thread-Safe Collections	Describes collection types such as System.Collections.Concurrent.BlockingCollection<T> and System.Collections.Concurrent.ConcurrentBag<T> that support safe and efficient concurrent access from multiple threads.
System.Collections.Immutable	Introduces the immutable collections and provides links to the collection types.

Reference

- System.Array
- System.Collections
- System.Collections.Concurrent
- System.Collections.Generic
- System.Collections.Specialized
- System.Linq
- System.Collections.Immutable