# Collections and Data Structures

Similar data can often be handled more efficiently when stored and manipulated as a collection. You can use the [System.Array](https://docs.microsoft.com/en-us/dotnet/api/system.array) class or the classes in the [System.Collections](https://docs.microsoft.com/en-us/dotnet/api/system.collections), [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic), [System.Collections.Concurrent](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent), and [System.Collections.Immutable](https://docs.microsoft.com/en-us/dotnet/api/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 were added in the .NET Framework 2.0 and provide collections that are type-safe at compile time. Because of this, generic collections typically offer better performance. Generic collections accept a type parameter when they are constructed and do not require that you cast to and from the [Object](https://docs.microsoft.com/en-us/dotnet/api/system.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](https://docs.microsoft.com/en-us/dotnet/api/system.object), require casting, and most are not supported for Windows Store app development. However, you may see non-generic collections in older code.

Starting with the .NET Framework 4, the collections in the [System.Collections.Concurrent](https://docs.microsoft.com/en-us/dotnet/api/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](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable) namespace ([NuGet package](https://www.nuget.org/packages/System.Collections.Immutable)) are inherently thread-safe because operations are performed on a copy of the original collection and the original collection cannot 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](https://docs.microsoft.com/en-us/dotnet/api/system.collections.icollection) interface or the [ICollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icollection-1) interface share these features:

* **The ability to enumerate the collection** .NET Framework collections either implement [System.Collections.IEnumerable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable) or [System.Collections.Generic.IEnumerable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1) 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](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/foreach-in) statement and the [For Each...Next Statement](https://docs.microsoft.com/en-us/dotnet/visual-basic/language-reference/statements/for-each-next-statement) use the enumerator exposed by the [GetEnumerator](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable.getenumerator) method and hide the complexity of manipulating the enumerator. In addition, any collection that implements [System.Collections.Generic.IEnumerable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1) is considered a *queryable type* and can be queried with LINQ. LINQ queries provide a common pattern for accessing data. They are 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#)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/linq-to-objects), [LINQ to Objects (Visual Basic)](https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/concepts/linq/linq-to-objects), [Parallel LINQ (PLINQ)](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq), [Introduction to LINQ Queries (C#)](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/introduction-to-linq-queries), and [Basic Query Operations (Visual Basic)](https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/concepts/linq/basic-query-operations).
* **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 reduces the code required to use the collection; however, the performance of the collection might be negatively affected. For example, for [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1), if [Count](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.count) is less than [Capacity](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.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](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.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](https://docs.microsoft.com/en-us/dotnet/api/system.collections.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](https://docs.microsoft.com/en-us/dotnet/api/system.collections) namespaces have a lower bound of zero, meaning they are 0-indexed. [Array](https://docs.microsoft.com/en-us/dotnet/api/system.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](https://docs.microsoft.com/en-us/dotnet/api/system.array.createinstance).
* **Synchronization for access from multiple threads** ([System.Collections](https://docs.microsoft.com/en-us/dotnet/api/system.collections) classes only).  
    
   Non-generic collection types in the [System.Collections](https://docs.microsoft.com/en-us/dotnet/api/system.collections) namespace provide some thread safety with synchronization; typically exposed through the [SyncRoot](https://docs.microsoft.com/en-us/dotnet/api/system.collections.icollection.syncroot) and [IsSynchronized](https://docs.microsoft.com/en-us/dotnet/api/system.collections.icollection.issynchronized) members. These collections are not 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](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent) namespace or consider using an immutable collection. For more information, see [Thread-Safe Collections](https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/).

## 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 are new to generic collections, this table will help you choose the generic collection that works the 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>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) | [Hashtable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable)  (A collection of key/value pairs that are organized based on the hash code of the key.) | [ConcurrentDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2)  [ReadOnlyDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.readonlydictionary-2)  [ImmutableDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutabledictionary-2) |
| Access items by index | [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) | [Array](https://docs.microsoft.com/en-us/dotnet/api/system.array)  [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist) | [ImmutableList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablelist-1)  [ImmutableArray](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablearray) |
| Use items first-in-first-out (FIFO) | [Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1) | [Queue](https://docs.microsoft.com/en-us/dotnet/api/system.collections.queue) | [ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1)  [ImmutableQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablequeue-1) |
| Use data Last-In-First-Out (LIFO) | [Stack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1) | [Stack](https://docs.microsoft.com/en-us/dotnet/api/system.collections.stack) | [ConcurrentStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1)  [ImmutableStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablestack-1) |
| Access items sequentially | [LinkedList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1) | No recommendation | No recommendation |
| Receive notifications when items are removed or added to the collection. (implements [INotifyPropertyChanged](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.inotifypropertychanged) and [INotifyCollectionChanged](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.inotifycollectionchanged)) | [ObservableCollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.observablecollection-1) | No recommendation | No recommendation |
| A sorted collection | [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2) | [SortedList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.sortedlist) | [ImmutableSortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesorteddictionary-2)  [ImmutableSortedSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesortedset-1) |
| A set for mathematical functions | [HashSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1)  [SortedSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedset-1) | No recommendation | [ImmutableHashSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablehashset-1)  [ImmutableSortedSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesortedset-1) |

### **Algorithmic complexity of collections**

When choosing a [collection class](https://docs.microsoft.com/en-us/dotnet/standard/collections/selecting-a-collection-class), it is 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 a simple array like List<T> uses. An array can be very 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>. That's because they both use binary trees. The significant difference, of course, is that ImmutableSortedSet<T> uses an immutable binary tree. Since ImmutableSortedSet<T> also offers a [System.Collections.Immutable.ImmutableSortedSet<T>.Builder](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesortedset-1.builder) class that allows mutation, you can have both immutability and performance.

## Related Topics

| Table 3 | |
| --- | --- |
| **Title** | **Description** |
| [Selecting a Collection Class](https://docs.microsoft.com/en-us/dotnet/standard/collections/selecting-a-collection-class) | Describes the different collections and helps you select one for your scenario. |
| [Commonly Used Collection Types](https://docs.microsoft.com/en-us/dotnet/standard/collections/commonly-used-collection-types) | Describes commonly used generic and nongeneric collection types such as [System.Array](https://docs.microsoft.com/en-us/dotnet/api/system.array), [System.Collections.Generic.List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1), and [System.Collections.Generic.Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2). |
| [When to Use Generic Collections](https://docs.microsoft.com/en-us/dotnet/standard/collections/when-to-use-generic-collections) | Discusses the use of generic collection types. |
| [Comparisons and Sorts Within Collections](https://docs.microsoft.com/en-us/dotnet/standard/collections/comparisons-and-sorts-within-collections) | Discusses the use of equality comparisons and sorting comparisons in collections. |
| [Sorted Collection Types](https://docs.microsoft.com/en-us/dotnet/standard/collections/sorted-collection-types) | Describes sorted collections performance and characteristics |
| [Hashtable and Dictionary Collection Types](https://docs.microsoft.com/en-us/dotnet/standard/collections/hashtable-and-dictionary-collection-types) | Describes the features of generic and non-generic hash-based dictionary types. |
| [Thread-Safe Collections](https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/) | Describes collection types such as [System.Collections.Concurrent.BlockingCollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1) and [System.Collections.Concurrent.ConcurrentBag<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1) 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](https://docs.microsoft.com/en-us/dotnet/api/system.array) [System.Collections](https://docs.microsoft.com/en-us/dotnet/api/system.collections) [System.Collections.Concurrent](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent) [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) [System.Collections.Specialized](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized) [System.Linq](https://docs.microsoft.com/en-us/dotnet/api/system.linq) [System.Collections.Immutable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable)

**Selecting a Collection Class**

Be sure to choose your collection class carefully. Using the wrong type can restrict your use of the collection.

Important

Avoid using the types in the [System.Collections](https://docs.microsoft.com/en-us/dotnet/api/system.collections) namespace. The generic and concurrent versions of the collections are recommended because of their greater type safety and other improvements.

Consider the following questions:

* Do you need a sequential list where the element is typically discarded after its value is retrieved?
  + If yes, consider using the [Queue](https://docs.microsoft.com/en-us/dotnet/api/system.collections.queue) class or the [Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1) generic class if you need first-in, first-out (FIFO) behavior. Consider using the [Stack](https://docs.microsoft.com/en-us/dotnet/api/system.collections.stack) class or the [Stack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1) generic class if you need last-in, first-out (LIFO) behavior. For safe access from multiple threads, use the concurrent versions, [ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1) and [ConcurrentStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1). For immutability, consider the immutable versions, [ImmutableQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablequeue-1) and [ImmutableStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablestack-1).
  + If not, consider using the other collections.
* Do you need to access the elements in a certain order, such as FIFO, LIFO, or random?
  + The [Queue](https://docs.microsoft.com/en-us/dotnet/api/system.collections.queue) class, as well as the [Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1), [ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1), and [ImmutableQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablequeue-1) generic classes all offer FIFO access. For more information, see [When to Use a Thread-Safe Collection](https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/when-to-use-a-thread-safe-collection).
  + The [Stack](https://docs.microsoft.com/en-us/dotnet/api/system.collections.stack) class, as well as the [Stack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1), [ConcurrentStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1), and [ImmutableStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablestack-1) generic classes all offer LIFO access. For more information, see [When to Use a Thread-Safe Collection](https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/when-to-use-a-thread-safe-collection).
  + The [LinkedList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1) generic class allows sequential access either from the head to the tail, or from the tail to the head.
* Do you need to access each element by index?
  + The [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist) and [StringCollection](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.stringcollection) classes and the [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) generic class offer access to their elements by the zero-based index of the element. For immutability, consider the immutable generic versions, [ImmutableArray<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablearray-1) and [ImmutableList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablelist-1).
  + The [Hashtable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable), [SortedList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.sortedlist), [ListDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.listdictionary), and [StringDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.stringdictionary) classes, and the [Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) and [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2) generic classes offer access to their elements by the key of the element. Additionally, there are immutable versions of several corresponding types: [ImmutableHashSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablehashset-1), [ImmutableDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutabledictionary-2), [ImmutableSortedSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesortedset-1), and [ImmutableSortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesorteddictionary-2).
  + The [NameObjectCollectionBase](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.nameobjectcollectionbase) and [NameValueCollection](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.namevaluecollection) classes, and the [KeyedCollection<TKey,TItem>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.keyedcollection-2) and [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2) generic classes offer access to their elements by either the zero-based index or the key of the element.
* Will each element contain one value, a combination of one key and one value, or a combination of one key and multiple values?
  + One value: Use any of the collections based on the [IList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ilist) interface or the [IList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ilist-1) generic interface. For an immutable option, consider the [IImmutableList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.iimmutablelist-1) generic interface.
  + One key and one value: Use any of the collections based on the [IDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.idictionary) interface or the [IDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.idictionary-2) generic interface. For an immutable option, consider the [IImmutableSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.iimmutableset-1) or [IImmutableDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.iimmutabledictionary-2) generic interfaces.
  + One value with embedded key: Use the [KeyedCollection<TKey,TItem>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.keyedcollection-2) generic class.
  + One key and multiple values: Use the [NameValueCollection](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.namevaluecollection) class.
* Do you need to sort the elements differently from how they were entered?
  + The [Hashtable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable) class sorts its elements by their hash codes.
  + The [SortedList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.sortedlist) class, and the [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2) and [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2) generic classes sort their elements by the key. The sort order is based on the implementation of the [IComparer](https://docs.microsoft.com/en-us/dotnet/api/system.collections.icomparer) interface for the [SortedList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.sortedlist) class and on the implementation of the [IComparer<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1) generic interface for the [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2) and [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2) generic classes. Of the two generic types, [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2) offers better performance than [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2), while [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2) consumes less memory.
  + [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist) provides a [Sort](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist.sort) method that takes an [IComparer](https://docs.microsoft.com/en-us/dotnet/api/system.collections.icomparer) implementation as a parameter. Its generic counterpart, the [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) generic class, provides a [Sort](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.sort) method that takes an implementation of the [IComparer<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1) generic interface as a parameter.
* Do you need fast searches and retrieval of information?
  + [ListDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.listdictionary) is faster than [Hashtable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable) for small collections (10 items or fewer). The [Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) generic class provides faster lookup than the [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2) generic class. The multi-threaded implementation is [ConcurrentDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2). [ConcurrentBag<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1) provides fast multi-threaded insertion for unordered data. For more information about both multi-threaded types, see [When to Use a Thread-Safe Collection](https://docs.microsoft.com/en-us/dotnet/standard/collections/thread-safe/when-to-use-a-thread-safe-collection).
* Do you need collections that accept only strings?
  + [StringCollection](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.stringcollection) (based on [IList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.ilist)) and [StringDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized.stringdictionary) (based on [IDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.idictionary)) are in the [System.Collections.Specialized](https://docs.microsoft.com/en-us/dotnet/api/system.collections.specialized) namespace.
  + In addition, you can use any of the generic collection classes in the [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) namespace as strongly typed string collections by specifying the [String](https://docs.microsoft.com/en-us/dotnet/api/system.string) class for their generic type arguments. For example, you can declare a variable to be of type [List<String>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) or [Dictionary<String,String>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2).

# When to Use a Thread-Safe Collection

The .NET Framework 4 introduces five new collection types that are specially designed to support multi-threaded add and remove operations. To achieve thread-safety, these new types use various kinds of efficient locking and lock-free synchronization mechanisms. Synchronization adds overhead to an operation. The amount of overhead depends on the kind of synchronization that is used, the kind of operations that are performed, and other factors such as the number of threads that are trying to concurrently access the collection.

In some scenarios, synchronization overhead is negligible and enables the multi-threaded type to perform significantly faster and scale far better than its non-thread-safe equivalent when protected by an external lock. In other scenarios, the overhead can cause the thread-safe type to perform and scale about the same or even more slowly than the externally-locked, non-thread-safe version of the type.

The following sections provide general guidance about when to use a thread-safe collection versus its non-thread-safe equivalent that has a user-provided lock around its read and write operations. Because performance may vary depending on many factors, the guidance is not specific and is not necessarily valid in all circumstances. If performance is very important, then the best way to determine which collection type to use is to measure performance based on representative computer configurations and loads. This document uses the following terms:

*Pure producer-consumer scenario*  
Any given thread is either adding or removing elements, but not both.

*Mixed producer-consumer scenario*  
Any given thread is both adding and removing elements.

*Speedup*  
Faster algorithmic performance relative to another type in the same scenario.

*Scalability*  
The increase in performance that is proportional to the number of cores on the computer. An algorithm that scales performs faster on eight cores than it does on two cores.

## **ConcurrentQueue(T) vs. Queue(T)**

In pure producer-consumer scenarios, where the processing time for each element is very small (a few instructions), then [System.Collections.Concurrent.ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1) can offer modest performance benefits over a [System.Collections.Generic.Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1) that has an external lock. In this scenario, [ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1) performs best when one dedicated thread is queuing and one dedicated thread is de-queuing. If you do not enforce this rule, then [Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1) might even perform slightly faster than [ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1) on computers that have multiple cores.

When processing time is around 500 FLOPS (floating point operations) or more, then the two-thread rule does not apply to [ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1), which then has very good scalability. [Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1) does not scale well in this scenario.

In mixed producer-consumer scenarios, when the processing time is very small, a [Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1) that has an external lock scales better than [ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1) does. However, when processing time is around 500 FLOPS or more, then the [ConcurrentQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentqueue-1) scales better.

## ConcurrentStack vs. Stack

In pure producer-consumer scenarios, when processing time is very small, then [System.Collections.Concurrent.ConcurrentStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1) and [System.Collections.Generic.Stack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1) that has an external lock will probably perform about the same with one dedicated pushing thread and one dedicated popping thread. However, as the number of threads increases, both types slow down because of increased contention, and [Stack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1) might perform better than [ConcurrentStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1). When processing time is around 500 FLOPS or more, then both types scale at about the same rate.

In mixed producer-consumer scenarios, [ConcurrentStack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1) is faster for both small and large workloads.

The use of the [PushRange](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1.pushrange) and [TryPopRange](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentstack-1.trypoprange) may greatly speed up access times.

## ConcurrentDictionary vs. Dictionary

In general, use a [System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2) in any scenario where you are adding and updating keys or values concurrently from multiple threads. In scenarios that involve frequent updates and relatively few reads, the [ConcurrentDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2) generally offers modest benefits. In scenarios that involve many reads and many updates, the [ConcurrentDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2) generally is significantly faster on computers that have any number of cores.

In scenarios that involve frequent updates, you can increase the degree of concurrency in the [ConcurrentDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2) and then measure to see whether performance increases on computers that have more cores. If you change the concurrency level, avoid global operations as much as possible.

If you are only reading key or values, the [Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) is faster because no synchronization is required if the dictionary is not being modified by any threads.

## ConcurrentBag

In pure producer-consumer scenarios, [System.Collections.Concurrent.ConcurrentBag<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1) will probably perform more slowly than the other concurrent collection types.

In mixed producer-consumer scenarios, [ConcurrentBag<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1) is generally much faster and more scalable than any other concurrent collection type for both large and small workloads.

## BlockingCollection

When bounding and blocking semantics are required, [System.Collections.Concurrent.BlockingCollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1) will probably perform faster than any custom implementation. It also supports rich cancellation, enumeration, and exception handling.