When to use generic collections

Article • 09/15/2021 • 3 minutes to read

Using generic collections gives you the automatic benefit of type safety without having to derive from a base collection type and implement type-specific members. Generic collection types also generally perform better than the corresponding nongeneric collection types (and better than types that are derived from nongeneric base collection types) when the collection elements are value types, because with generics, there's no need to box the elements.

For programs that target .NET Standard 1.0 or later, use the generic collection classes in the System.Collections.Concurrent namespace when multiple threads might be adding or removing items from the collection concurrently. Additionally, when immutability is desired, consider the generic collection classes in the System.Collections.Immutable namespace.

The following generic types correspond to existing collection types:

- List<T> is the generic class that corresponds to ArrayList.
- Dictionary<TKey,TValue> and ConcurrentDictionary<TKey,TValue> are the generic classes that correspond to Hashtable.
- Collection<T> is the generic class that corresponds to CollectionBase. Collection<T> can be used as a base class, but unlike CollectionBase, it is not abstract, which makes it much easier to use.

it much easier to use.

- ReadOnlyCollection<T> is the generic class that corresponds to ReadOnlyCollectionBase. ReadOnlyCollection<T> is not abstract and has a constructor that makes it easy to expose an existing List<T> as a read-only collection.
- The Queue<T>, ConcurrentQueue<T>, ImmutableQueue<T>, ImmutableArray<T>, SortedList<TKey,TValue>, and ImmutableSortedSet<T> generic classes correspond to the respective nongeneric classes with the same names.

Additional Types

Several generic collection types do not have nongeneric counterparts. They include the following:

- LinkedList<T> is a general-purpose linked list that provides O(1) insertion and removal operations.
- SortedDictionary<TKey,TValue> is a sorted dictionary with O(log `n`) insertion and retrieval operations, which makes it a useful alternative to SortedList<TKey,TValue>.
- KeyedCollection<TKey,TItem> is a hybrid between a list and a dictionary, which provides a way to store objects that contain their own keys.
- BlockingCollection<T> implements a collection class with bounding and blocking functionality.
- ConcurrentBag<T> provides fast insertion and removal of unordered elements.

Immutable builders

When you desire immutability functionality in your app, the System.Collections.Immutable namespace offers generic collection types you can use. All of the immutable collection types offer `Builder` classes that can optimize performance when you're performing multiple mutations. The `Builder` class batches operations in a mutable state. When all mutations have been completed, call the `ToImmutable` method to "freeze" all nodes and create an immutable generic collection, for example, an ImmutableList<T>.

The `Builder` object can be created by calling the nongeneric `CreateBuilder()` method. From a `Builder` instance, you can call `ToImmutable()`. Likewise, from the `Immutable*` collection, you can call `ToBuilder()` to create a builder instance from the generic

immutable collection. The following are the various `Builder` types.

- ImmutableArray<T>.Builder
- ImmutableDictionary<TKey,TValue>.Builder
- ImmutableHashSet<T>.Builder
- ImmutableList<T>.Builder
- ImmutableSortedDictionary<TKey,TValue>.Builder
- ImmutableSortedSet<T>.Builder

LINQ to Objects

The LINQ to Objects feature enables you to use LINQ queries to access in-memory objects as long as the object type implements the System.Collections.IEnumerable or

System.Collections.Generic.IEnumerable<T> interface. LINQ queries provide a common pattern for accessing data; 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#), LINQ to Objects (Visual Basic), and Parallel LINQ (PLINQ).

Additional Functionality

Some of the generic types have functionality that is not found in the nongeneric collection types. For example, the List<T> class, which corresponds to the nongeneric ArrayList class, has a number of methods that accept generic delegates, such as the Predicate<T> delegate that allows you to specify methods for searching the list, the Action<T> delegate that represents methods that act on each element of the list, and the Converter<TInput,TOutput> delegate that lets you define conversions between types.

The List<T> class allows you to specify your own IComparer<T> generic interface implementations for sorting and searching the list. The SortedDictionary<TKey,TValue> and SortedList<TKey,TValue> classes also have this capability. In addition, these classes let you specify comparers when the collection is created. In similar fashion, the Dictionary<TKey,TValue> and KeyedCollection<TKey,TItem> classes let you specify your own equality comparers.

See also