**Generics in .NET**

* 20minutes to read

Generics let you tailor a method, class, structure, or interface to the precise data type it acts upon. For example, instead of using the [Hashtable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable) class, which allows keys and values to be of any type, you can use the [Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) generic class and specify the types allowed for the key and the value. Among the benefits of generics are increased code reusability and type safety.

**Defining and Using Generics**

Generics are classes, structures, interfaces, and methods that have placeholders (type parameters) for one or more of the types that they store or use. A generic collection class might use a type parameter as a placeholder for the type of objects that it stores; the type parameters appear as the types of its fields and the parameter types of its methods. A generic method might use its type parameter as the type of its return value or as the type of one of its formal parameters. The following code illustrates a simple generic class definition.

C#Copy

public class Generic<T>

{

public T Field;

}

When you create an instance of a generic class, you specify the actual types to substitute for the type parameters. This establishes a new generic class, referred to as a constructed generic class, with your chosen types substituted everywhere that the type parameters appear. The result is a type-safe class that is tailored to your choice of types, as the following code illustrates.

C#Copy

public static void Main()

{

Generic<string> g = new Generic<string>();

g.Field = "A string";

//...

Console.WriteLine("Generic.Field = \"{0}\"", g.Field);

Console.WriteLine("Generic.Field.GetType() = {0}", g.Field.GetType().FullName);

}

**Generics terminology**

The following terms are used to discuss generics in .NET:

* A *generic type definition* is a class, structure, or interface declaration that functions as a template, with placeholders for the types that it can contain or use. For example, the [System.Collections.Generic.Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) class can contain two types: keys and values. Because a generic type definition is only a template, you cannot create instances of a class, structure, or interface that is a generic type definition.
* *Generic type parameters*, or *type parameters*, are the placeholders in a generic type or method definition. The [System.Collections.Generic.Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) generic type has two type parameters, TKey and TValue, that represent the types of its keys and values.
* A *constructed generic type*, or *constructed type*, is the result of specifying types for the generic type parameters of a generic type definition.
* A *generic type argument* is any type that is substituted for a generic type parameter.
* The general term *generic type* includes both constructed types and generic type definitions.
* *Covariance* and *contravariance* of generic type parameters enable you to use constructed generic types whose type arguments are more derived (covariance) or less derived (contravariance) than a target constructed type. Covariance and contravariance are collectively referred to as *variance*. For more information, see [Covariance and Contravariance](https://docs.microsoft.com/en-us/dotnet/standard/generics/covariance-and-contravariance).
* *Constraints* are limits placed on generic type parameters. For example, you might limit a type parameter to types that implement the [System.Collections.Generic.IComparer<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1) generic interface, to ensure that instances of the type can be ordered. You can also constrain type parameters to types that have a particular base class, that have a parameterless constructor, or that are reference types or value types. Users of the generic type cannot substitute type arguments that do not satisfy the constraints.
* A *generic method definition* is a method with two parameter lists: a list of generic type parameters and a list of formal parameters. Type parameters can appear as the return type or as the types of the formal parameters, as the following code shows.

C#Copy

T Generic<T>(T arg)

{

T temp = arg;

//...

return temp;

}

Generic methods can appear on generic or nongeneric types. It is important to note that a method is not generic just because it belongs to a generic type, or even because it has formal parameters whose types are the generic parameters of the enclosing type. A method is generic only if it has its own list of type parameters. In the following code, only method G is generic.

C#Copy

class A

{

T G<T>(T arg)

{

T temp = arg;

//...

return temp;

}

}

class Generic<T>

{

T M(T arg)

{

T temp = arg;

//...

return temp;

}

}

**Advantages and disadvantages of generics**

There are many advantages to using generic collections and delegates:

* Type safety. Generics shift the burden of type safety from you to the compiler. There is no need to write code to test for the correct data type because it is enforced at compile time. The need for type casting and the possibility of run-time errors are reduced.
* Less code and code is more easily reused. There is no need to inherit from a base type and override members. For example, the [LinkedList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1) is ready for immediate use. For example, you can create a linked list of strings with the following variable declaration:

C#Copy

LinkedList<string> llist = new LinkedList<string>();

* Better performance. Generic collection types generally perform better for storing and manipulating value types because there is no need to box the value types.
* Generic delegates enable type-safe callbacks without the need to create multiple delegate classes. For example, the [Predicate<T>](https://docs.microsoft.com/en-us/dotnet/api/system.predicate-1) generic delegate allows you to create a method that implements your own search criteria for a particular type and to use your method with methods of the [Array](https://docs.microsoft.com/en-us/dotnet/api/system.array) type such as [Find](https://docs.microsoft.com/en-us/dotnet/api/system.array.find), [FindLast](https://docs.microsoft.com/en-us/dotnet/api/system.array.findlast), and [FindAll](https://docs.microsoft.com/en-us/dotnet/api/system.array.findall).
* Generics streamline dynamically generated code. When you use generics with dynamically generated code you do not need to generate the type. This increases the number of scenarios in which you can use lightweight dynamic methods instead of generating entire assemblies. For more information, see [How to: Define and Execute Dynamic Methods](https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/how-to-define-and-execute-dynamic-methods) and [DynamicMethod](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.dynamicmethod).

The following are some limitations of generics:

* Generic types can be derived from most base classes, such as [MarshalByRefObject](https://docs.microsoft.com/en-us/dotnet/api/system.marshalbyrefobject) (and constraints can be used to require that generic type parameters derive from base classes like [MarshalByRefObject](https://docs.microsoft.com/en-us/dotnet/api/system.marshalbyrefobject)). However, .NET does not support context-bound generic types. A generic type can be derived from [ContextBoundObject](https://docs.microsoft.com/en-us/dotnet/api/system.contextboundobject), but trying to create an instance of that type causes a [TypeLoadException](https://docs.microsoft.com/en-us/dotnet/api/system.typeloadexception).
* Enumerations cannot have generic type parameters. An enumeration can be generic only incidentally (for example, because it is nested in a generic type that is defined using Visual Basic, C#, or C++). For more information, see "Enumerations" in [Common Type System](https://docs.microsoft.com/en-us/dotnet/standard/base-types/common-type-system).
* Lightweight dynamic methods cannot be generic.
* In Visual Basic, C#, and C++, a nested type that is enclosed in a generic type cannot be instantiated unless types have been assigned to the type parameters of all enclosing types. Another way of saying this is that in reflection, a nested type that is defined using these languages includes the type parameters of all its enclosing types. This allows the type parameters of enclosing types to be used in the member definitions of a nested type. For more information, see "Nested Types" in [MakeGenericType](https://docs.microsoft.com/en-us/dotnet/api/system.type.makegenerictype).

**Note**

A nested type that is defined by emitting code in a dynamic assembly or by using the [**Ilasm.exe (IL Assembler)**](https://docs.microsoft.com/en-us/dotnet/framework/tools/ilasm-exe-il-assembler) is not required to include the type parameters of its enclosing types; however, if it does not include them, the type parameters are not in scope in the nested class.

For more information, see "Nested Types" in [MakeGenericType](https://docs.microsoft.com/en-us/dotnet/api/system.type.makegenerictype).

**Class Library and Language Support**

.NET provides a number of generic collection classes in the following namespaces:

* The [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) namespace contains most of the generic collection types provided by .NET, such as the [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) and [Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) generic classes.
* The [System.Collections.ObjectModel](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel) namespace contains additional generic collection types, such as the [ReadOnlyCollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.readonlycollection-1) generic class, that are useful for exposing object models to users of your classes.

Generic interfaces for implementing sort and equality comparisons are provided in the [System](https://docs.microsoft.com/en-us/dotnet/api/system) namespace, along with generic delegate types for event handlers, conversions, and search predicates.

Support for generics has been added to the [System.Reflection](https://docs.microsoft.com/en-us/dotnet/api/system.reflection) namespace for examining generic types and generic methods, to [System.Reflection.Emit](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit) for emitting dynamic assemblies that contain generic types and methods, and to [System.CodeDom](https://docs.microsoft.com/en-us/dotnet/api/system.codedom) for generating source graphs that include generics.

The common language runtime provides new opcodes and prefixes to support generic types in Microsoft intermediate language (MSIL), including [Stelem](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.stelem), [Ldelem](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.ldelem), [Unbox\_Any](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.unbox_any), [Constrained](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.constrained), and [Readonly](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes.readonly).

Visual C++, C#, and Visual Basic all provide full support for defining and using generics. For more information about language support, see [Generic Types in Visual Basic](https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/data-types/generic-types), [Introduction to Generics](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics), and [Overview of Generics in Visual C++](https://docs.microsoft.com/en-us/cpp/windows/overview-of-generics-in-visual-cpp).

**Nested Types and Generics**

A type that is nested in a generic type can depend on the type parameters of the enclosing generic type. The common language runtime considers nested types to be generic, even if they do not have generic type parameters of their own. When you create an instance of a nested type, you must specify type arguments for all enclosing generic types.

**Related Topics**

| **RELATED TOPICS** | |
| --- | --- |
| **Title** | **Description** |
| [Generic Collections in .NET](https://docs.microsoft.com/en-us/dotnet/standard/generics/collections) | Describes generic collection classes and other generic types in .NET. |
| [Generic Delegates for Manipulating Arrays and Lists](https://docs.microsoft.com/en-us/dotnet/standard/generics/delegates-for-manipulating-arrays-and-lists) | Describes generic delegates for conversions, search predicates, and actions to be taken on elements of an array or collection. |
| [Generic Interfaces](https://docs.microsoft.com/en-us/dotnet/standard/generics/interfaces) | Describes generic interfaces that provide common functionality across families of generic types. |
| [Covariance and Contravariance](https://docs.microsoft.com/en-us/dotnet/standard/generics/covariance-and-contravariance) | Describes covariance and contravariance in generic type parameters. |
| [Commonly Used Collection Types](https://docs.microsoft.com/en-us/dotnet/standard/collections/commonly-used-collection-types) | Provides summary information about the characteristics and usage scenarios of the collection types in .NET, including generic types. |
| [When to Use Generic Collections](https://docs.microsoft.com/en-us/dotnet/standard/collections/when-to-use-generic-collections) | Describes general rules for determining when to use generic collection types. |
| [How to: Define a Generic Type with Reflection Emit](https://docs.microsoft.com/en-us/dotnet/framework/reflection-and-codedom/how-to-define-a-generic-type-with-reflection-emit) | Explains how to generate dynamic assemblies that include generic types and methods. |
| [Generic Types in Visual Basic](https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/data-types/generic-types) | Describes the generics feature for Visual Basic users, including how-to topics for using and defining generic types. |
| [Introduction to Generics](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics) | Provides an overview of defining and using generic types for C# users. |
| [Overview of Generics in Visual C++](https://docs.microsoft.com/en-us/cpp/windows/overview-of-generics-in-visual-cpp) | Describes the generics feature for C++ users, including the differences between generics and templates. |

**Reference**

[System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic)

[System.Collections.ObjectModel](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel)

[System.Reflection.Emit.OpCodes](https://docs.microsoft.com/en-us/dotnet/api/system.reflection.emit.opcodes)

**Recommended content**

**[Collections and Data Structures](https://docs.microsoft.com/en-us/dotnet/standard/collections/)**

Learn how to use collections and data structures in .NET. Use generic and non-generic collections in thread-safe operations.

**[When to Use Generic Collections](https://docs.microsoft.com/en-us/dotnet/standard/collections/when-to-use-generic-collections)**

Learn more about: When to use generic collections

**[Generics - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/)**

Learn about generics. Generic types maximize code reuse, type safety, and performance, and are commonly used to create collection classes.

**[Boxing and Unboxing - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing)**

Learn about boxing and unboxing in C# programming. See code examples and view additional available resources.

**[Value types - C# reference](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/value-types)**

Learn about value types, its kinds, and the built-in ones in C#

**[Indexers - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/)**

Indexers in C# allow class or struct instances to be indexed like arrays. You can set or get the indexed value without specifying a type or instance member.

**[Abstract and Sealed Classes and Class Members - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/abstract-and-sealed-classes-and-class-members)**

The abstract keyword in C# creates incomplete classes and class members. The sealed keyword prevents inheritance of previously virtual classes or class members.

**[Polymorphism - C# Programming Guide](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism)**

Learn about polymorphism, a key concept in object-oriented programming languages like C#, which describes the relationship between base and derived classes.

Show less

**Generic types overview**

Developers use generics all the time in .NET, whether implicitly or explicitly. When you use LINQ in .NET, did you ever notice that you're working with [IEnumerable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1)? Or if you ever saw an online sample of a "generic repository" for talking to databases using Entity Framework, did you see that most methods return IQueryable<T>? You may have wondered what the **T** is in these examples and why it's in there.

First introduced in the .NET Framework 2.0, generics are essentially a "code template" that allows developers to define [type-safe](https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-4.0/hbzz1a9a(v=vs.100)) data structures without committing to an actual data type. For example, [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) is a [generic collection](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) that can be declared and used with any type, such as List<int>, List<string>, or List<Person>.

To understand why generics are useful, let's take a look at a specific class before and after adding generics: [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist). In .NET Framework 1.0, the ArrayList elements were of type [Object](https://docs.microsoft.com/en-us/dotnet/api/system.object). Any element added to the collection was silently converted into an Object. The same would happen when reading elements from the list. This process is known as [boxing and unboxing](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/boxing-and-unboxing), and it impacts performance. Aside from performance, however, there's no way to determine the type of data in the list at compile time, which makes for some fragile code. Generics solve this problem by defining the type of data each instance of list will contain. For example, you can only add integers to List<int> and only add Persons to List<Person>.

Generics are also available at run time. The runtime knows what type of data structure you're using and can store it in memory more efficiently.

The following example is a small program that illustrates the efficiency of knowing the data structure type at run time:

C#Copy

using System;

using System.Collections;

using System.Collections.Generic;

using System.Diagnostics;

namespace GenericsExample {

class Program {

static void Main(string[] args) {

//generic list

List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };

//non-generic list

ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };

// timer for generic list sort

Stopwatch s = Stopwatch.StartNew();

ListGeneric.Sort();

s.Stop();

Console.WriteLine($"Generic Sort: {ListGeneric} \n Time taken: {s.Elapsed.TotalMilliseconds}ms");

//timer for non-generic list sort

Stopwatch s2 = Stopwatch.StartNew();

ListNonGeneric.Sort();

s2.Stop();

Console.WriteLine($"Non-Generic Sort: {ListNonGeneric} \n Time taken: {s2.Elapsed.TotalMilliseconds}ms");

Console.ReadLine();

}

}

}

This program produces output similar to the following:

ConsoleCopy

Generic Sort: System.Collections.Generic.List`1[System.Int32]

Time taken: 0.0034ms

Non-Generic Sort: System.Collections.ArrayList

Time taken: 0.2592ms

The first thing you can notice here is that sorting the generic list is significantly faster than sorting the non-generic list. You might also notice that the type for the generic list is distinct ([System.Int32]), whereas the type for the non-generic list is generalized. Because the runtime knows the generic List<int> is of type [Int32](https://docs.microsoft.com/en-us/dotnet/api/system.int32), it can store the list elements in an underlying integer array in memory, while the non-generic ArrayList has to cast each list element to an object. As this example shows, the extra casts take up time and slow down the list sort.

An additional advantage of the runtime knowing the type of your generic is a better debugging experience. When you're debugging a generic in C#, you know what type each element is in your data structure. Without generics, you would have no idea what type each element was.

**Generic collections in .NET**

* 2 minutes to read

The .NET class library provides a number of generic collection classes in the [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic) and [System.Collections.ObjectModel](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel) namespaces. For more detailed information about these classes, see [Commonly Used Collection Types](https://docs.microsoft.com/en-us/dotnet/standard/collections/commonly-used-collection-types).

**System.Collections.Generic**

Many of the generic collection types are direct analogs of nongeneric types. [Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) is a generic version of [Hashtable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable); it uses the generic structure [KeyValuePair<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.keyvaluepair-2) for enumeration instead of [DictionaryEntry](https://docs.microsoft.com/en-us/dotnet/api/system.collections.dictionaryentry).

[List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) is a generic version of [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist). There are generic [Queue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.queue-1) and [Stack<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.stack-1) classes that correspond to the nongeneric versions.

There are generic and nongeneric versions of [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2). Both versions are hybrids of a dictionary and a list. The [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2) generic class is a pure dictionary and has no nongeneric counterpart.

The [LinkedList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1) generic class is a true linked list. It has no nongeneric counterpart.

**System.Collections.ObjectModel**

The [Collection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.collection-1) generic class provides a base class for deriving your own generic collection types. The [ReadOnlyCollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.readonlycollection-1) class provides an easy way to produce a read-only collection from any type that implements the [IList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ilist-1) generic interface. The [KeyedCollection<TKey,TItem>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.keyedcollection-2) generic class provides a way to store objects that contain their own keys.

**Other generic types**

The [Nullable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.nullable-1) generic structure allows you to use value types as if they could be assigned null. This can be useful when working with database queries, where fields that contain value types can be missing. The generic type parameter can be any value type.

**Note**

In C# and Visual Basic, it is not necessary to use [**Nullable<T>**](https://docs.microsoft.com/en-us/dotnet/api/system.nullable-1) explicitly because the language has syntax for nullable types. See [**Nullable value types (C# reference)**](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nullable-value-types) and [**Nullable value types (Visual Basic)**](https://docs.microsoft.com/en-us/dotnet/visual-basic/programming-guide/language-features/data-types/nullable-value-types).

The [ArraySegment<T>](https://docs.microsoft.com/en-us/dotnet/api/system.arraysegment-1) generic structure provides a way to delimit a range of elements within a one-dimensional, zero-based array of any type. The generic type parameter is the type of the array's elements.

The [EventHandler<TEventArgs>](https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler-1) generic delegate eliminates the need to declare a delegate type to handle events, if your event follows the event-handling pattern used by .NET. For example, suppose you have created a MyEventArgs class, derived from [EventArgs](https://docs.microsoft.com/en-us/dotnet/api/system.eventargs), to hold the data for your event. You can then declare the event as follows:

C#Copy

public event EventHandler<MyEventArgs> MyEvent;

**See also**

* [System.Collections.Generic](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic)
* [System.Collections.ObjectModel](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel)
* [Generics](https://docs.microsoft.com/en-us/dotnet/standard/generics/)
* [Generic Delegates for Manipulating Arrays and Lists](https://docs.microsoft.com/en-us/dotnet/standard/generics/delegates-for-manipulating-arrays-and-lists)
* [Generic Interfaces](https://docs.microsoft.com/en-us/dotnet/standard/generics/interfaces)

**When to use generic collections**

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](https://docs.microsoft.com/en-us/dotnet/api/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](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable) namespace.

The following generic types correspond to existing collection types:

* [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) is the generic class that corresponds to [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist).
* [Dictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) and [ConcurrentDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2) are the generic classes that correspond to [Hashtable](https://docs.microsoft.com/en-us/dotnet/api/system.collections.hashtable).
* [Collection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.collection-1) is the generic class that corresponds to [CollectionBase](https://docs.microsoft.com/en-us/dotnet/api/system.collections.collectionbase). [Collection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.collection-1) can be used as a base class, but unlike [CollectionBase](https://docs.microsoft.com/en-us/dotnet/api/system.collections.collectionbase), it is not abstract, which makes it much easier to use.
* [ReadOnlyCollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.readonlycollection-1) is the generic class that corresponds to [ReadOnlyCollectionBase](https://docs.microsoft.com/en-us/dotnet/api/system.collections.readonlycollectionbase). [ReadOnlyCollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.readonlycollection-1) is not abstract and has a constructor that makes it easy to expose an existing [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) as a read-only collection.
* 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), [ImmutableQueue<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablequeue-1), [ImmutableArray<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablearray-1), [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2), and [ImmutableSortedSet<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesortedset-1) 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>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1) is a general-purpose linked list that provides O(1) insertion and removal operations.
* [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2) is a sorted dictionary with O(log n) insertion and retrieval operations, which makes it a useful alternative to [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2).
* [KeyedCollection<TKey,TItem>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.keyedcollection-2) is a hybrid between a list and a dictionary, which provides a way to store objects that contain their own keys.
* [BlockingCollection<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.blockingcollection-1) implements a collection class with bounding and blocking functionality.
* [ConcurrentBag<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentbag-1) provides fast insertion and removal of unordered elements.

**Immutable builders**

When you desire immutability functionality in your app, the [System.Collections.Immutable](https://docs.microsoft.com/en-us/dotnet/api/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>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablelist-1).

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](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablearray-1.builder)
* [ImmutableDictionary<TKey,TValue>.Builder](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutabledictionary-2.builder)
* [ImmutableHashSet<T>.Builder](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablehashset-1.builder)
* [ImmutableList<T>.Builder](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablelist-1.builder)
* [ImmutableSortedDictionary<TKey,TValue>.Builder](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesorteddictionary-2.builder)
* [ImmutableSortedSet<T>.Builder](https://docs.microsoft.com/en-us/dotnet/api/system.collections.immutable.immutablesortedset-1.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](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) 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#)](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), and [Parallel LINQ (PLINQ)](https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq).

**Additional Functionality**

Some of the generic types have functionality that is not found in the nongeneric collection types. For example, the [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) class, which corresponds to the nongeneric [ArrayList](https://docs.microsoft.com/en-us/dotnet/api/system.collections.arraylist) class, has a number of methods that accept generic delegates, such as the [Predicate<T>](https://docs.microsoft.com/en-us/dotnet/api/system.predicate-1) delegate that allows you to specify methods for searching the list, the [Action<T>](https://docs.microsoft.com/en-us/dotnet/api/system.action-1) delegate that represents methods that act on each element of the list, and the [Converter<TInput,TOutput>](https://docs.microsoft.com/en-us/dotnet/api/system.converter-2) delegate that lets you define conversions between types.

The [List<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) class allows you to specify your own [IComparer<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1) generic interface implementations for sorting and searching the list. The [SortedDictionary<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2) and [SortedList<TKey,TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedlist-2) 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>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2) and [KeyedCollection<TKey,TItem>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.keyedcollection-2) classes let you specify your own equality comparers.