Collections (C#)

Visual Studio 2015

For many applications, you want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects.

Arrays are most useful for creating and working with a fixed number of strongly-typed objects. For information about arrays, see Arrays (C# Programming Guide).

Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.

A collection is a class, so you must declare an instance of the class before you can add elements to that collection.

If your collection contains elements of only one data type, you can use one of the classes in the System.Collections.Generic namespace. A generic collection enforces type safety so that no other data type can be added to it. When you retrieve an element from a generic collection, you do not have to determine its data type or convert it.

Note

For the examples in this topic, includeusing directives for the **System.Collections.Generic** and **System.Linq** namespaces.

In this topic

- Iterators

Using a Simple Collection

The examples in this section use the generic List<T> class, which enables you to work with a strongly typed list of objects.

The following example creates a list of strings and then iterates through the strings by using a or foreach statement.

C#

```csharp
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

If the contents of a collection are known in advance, you can use a *collection initializer* to initialize the collection. For more information, see Object and Collection Initializers (C# Programming Guide).

The following example is the same as the previous example, except a collection initializer is used to add elements to the collection.

C#

```csharp
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

You can use a for statement instead of a **foreach** statement to iterate through a collection. You accomplish this by accessing the collection elements by the index position. The index of the elements starts at 0 and ends at the element count minus 1.

The following example iterates through the elements of a collection by using **for** instead of **foreach**.

C#

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

The following example removes an element from the collection by specifying the object to remove.

C#

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

The following example removes elements from a generic list. Instead of a **foreach** statement, a for statement that iterates in descending order is used. This is because the RemoveAt method causes elements after a removed element to have a lower index value.

C#

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

For the type of elements in the List<T>, you can also define your own class. In the following example, the `Galaxy` class that is used by the List<T> is defined in the code.

C#

```csharp
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
        {
            new Galaxy() { Name="Tadpole", MegaLightYears=400},
            new Galaxy() { Name="Pinwheel", MegaLightYears=25},
            new Galaxy() { Name="Milky Way", MegaLightYears=0},
            new Galaxy() { Name="Andromeda", MegaLightYears=3}
        };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

Kinds of Collections

Many common collections are provided by the .NET Framework. For a complete list, see System.Collections namespaces. Each type of collection is designed for a specific purpose.

Some of the common collection classes are described in this section:

- **System.Collections.Generic** classes
- **System.Collections.Concurrent** classes
- **System.Collections** classes

System.Collections.Generic Classes

You can create a generic collection by using one of the classes in the System.Collections.Generic namespace. A generic collection is useful when every item in the collection has the same data type. A generic collection enforces strong typing by allowing only the desired data type to be added.

The following table lists some of the frequently used classes of the System.Collections.Generic namespace:

Class	Description
Dictionary<TKey, TValue>	Represents a collection of key/value pairs that are organized based on the key.
List<T>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey, TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

For additional information, see Commonly Used Collection Types, Selecting a Collection Class, and System.Collections.Generic.

System.Collections.Concurrent Classes

In the .NET Framework 4 or newer, the collections in the System.Collections.Concurrent namespace provide efficient thread-safe operations for accessing collection items from multiple threads.

The classes in the System.Collections.Concurrent namespace should be used instead of the corresponding types in the System.Collections.Generic and System.Collections namespaces whenever multiple threads are accessing the collection concurrently. For more information, see Thread-Safe Collections and System.Collections.Concurrent.

Some classes included in the System.Collections.Concurrent namespace are BlockingCollection<T>, ConcurrentDictionary<TKey, TValue>, ConcurrentQueue<T>, and ConcurrentStack<T>.

System.Collections Classes

The classes in the System.Collections namespace do not store elements as specifically typed objects, but as objects of type **Object**.

Whenever possible, you should use the generic collections in the System.Collections.Generic namespace or the System.Collections.Concurrent namespace instead of the legacy types in the **System.Collections** namespace.

The following table lists some of the frequently used classes in the **System.Collections** namespace:

Class	Description
ArrayList	Represents an array of objects whose size is dynamically increased as required.
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.
Queue	Represents a first in, first out (FIFO) collection of objects.
Stack	Represents a last in, first out (LIFO) collection of objects.

The System.Collections.Specialized namespace provides specialized and strongly typed collection classes, such as string-only collections and linked-list and hybrid dictionaries.

Implementing a Collection of Key/Value Pairs

The Dictionary<TKey, TValue> generic collection enables you to access to elements in a collection by using the key of each element. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is fast because the **Dictionary** class is implemented as a hash table.

The following example creates a **Dictionary** collection and iterates through the dictionary by using a **foreach** statement.

C#

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
            Element theElement = kvp.Value;

            Console.WriteLine("key: " + kvp.Key);
            Console.WriteLine("values: " + theElement.Symbol + " " +
                theElement.Name + " " + theElement.AtomicNumber);
        }
    }

    private static Dictionary<string, Element> BuildDictionary()
    {
        var elements = new Dictionary<string, Element>();

        AddToDictionary(elements, "K", "Potassium", 19);
        AddToDictionary(elements, "Ca", "Calcium", 20);
        AddToDictionary(elements, "Sc", "Scandium", 21);
        AddToDictionary(elements, "Ti", "Titanium", 22);

        return elements;
    }

    private static void AddToDictionary(Dictionary<string, Element> elements,
        string symbol, string name, int atomicNumber)
    {
        Element theElement = new Element();

        theElement.Symbol = symbol;
        theElement.Name = name;
        theElement.AtomicNumber = atomicNumber;

        elements.Add(key: theElement.Symbol, value: theElement);
    }

    public class Element
    {
        public string Symbol { get; set; }
        public string Name { get; set; }
        public int AtomicNumber { get; set; }
    }
```

To instead use a collection initializer to build the **Dictionary** collection, you can replace the `BuildDictionary` and `AddToDictionary` methods with the following method.

C#

```
    private static Dictionary<string, Element> BuildDictionary2()
    {
        return new Dictionary<string, Element>
        {
            {"K",
                new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
            {"Ca",
            new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}
```

The following example uses the ContainsKey method and the Item property of **Dictionary** to quickly find an item by key. The **Item** property enables you to access an item in the `elements` collection by using the `elements[symbol]` in C#.

C#

```
private static void FindInDictionary(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    if (elements.ContainsKey(symbol) == false)
    {
        Console.WriteLine(symbol + " not found");
    }
    else
    {
        Element theElement = elements[symbol];
        Console.WriteLine("found: " + theElement.Name);
    }
}
```

The following example instead uses the TryGetValue method quickly find an item by key.

C#

```
private static void FindInDictionary2(string symbol)
{
    Dictionary<string, Element> elements = BuildDictionary();

    Element theElement = null;
    if (elements.TryGetValue(symbol, out theElement) == false)
        Console.WriteLine(symbol + " not found");
    else
        Console.WriteLine("found: " + theElement.Name);
}
```

Using LINQ to Access a Collection

LINQ (Language-Integrated Query) can be used to access collections. LINQ queries provide filtering, ordering, and grouping capabilities. For more information, see Getting Started with LINQ in C#.

The following example runs a LINQ query against a generic **List**. The LINQ query returns a different collection that contains the results.

C#

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList()
{
    return new List<Element>
    {
        { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
}

public class Element
{
    public string Symbol { get; set; }
    public string Name { get; set; }
    public int AtomicNumber { get; set; }
}
```

Sorting a Collection

The following example illustrates a procedure for sorting a collection. The example sorts instances of the `Car` class that are stored in a List<T>. The `Car` class implements the IComparable<T> interface, which requires that the CompareTo method be implemented.

Each call to the CompareTo method makes a single comparison that is used for sorting. User-written code in the **CompareTo** method returns a value for each comparison of the current object with another object. The value returned is less than zero if the current object is less than the other object, greater than zero if the current object is greater than the other object, and zero if they are equal. This enables you to define in code the criteria for greater than, less than, and equal.

In the `ListCars` method, the `cars.Sort()` statement sorts the list. This call to the Sort method of the List<T> causes the **CompareTo** method to be called automatically for the `Car` objects in the **List**.

C#

```csharp
private static void ListCars()
{
    var cars = new List<Car>
    {
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.Write(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    //  blue  50 car4
    //  blue  30 car5
    //  blue  20 car1
    //  green 50 car7
    //  green 10 car3
    //  red   60 car6
```

```
        //  red   50 car2
    }

    public class Car : IComparable<Car>
    {
        public string Name { get; set; }
        public int Speed { get; set; }
        public string Color { get; set; }

        public int CompareTo(Car other)
        {
            // A call to this method makes a single comparison that is
            // used for sorting.

            // Determine the relative order of the objects being compared.
            // Sort by color alphabetically, and then by speed in
            // descending order.

            // Compare the colors.
            int compare;
            compare = String.Compare(this.Color, other.Color, true);

            // If the colors are the same, compare the speeds.
            if (compare == 0)
            {
                compare = this.Speed.CompareTo(other.Speed);

                // Use descending order for speed.
                compare = -compare;
            }

            return compare;
        }
    }
```

Defining a Custom Collection

You can define a collection by implementing the IEnumerable<T> or IEnumerable interface. For additional information, see How to: Access a Collection Class with foreach (C# Programming Guide).

Although you can define a custom collection, it is usually better to instead use the collections that are included in the .NET Framework, which are described in Kinds of Collections earlier in this topic.

The following example defines a custom collection class named `AllColors`. This class implements the IEnumerable interface, which requires that the GetEnumerator method be implemented.

The **GetEnumerator** method returns an instance of the `ColorEnumerator` class. `ColorEnumerator` implements the IEnumerator interface, which requires that the Current property, MoveNext method, and Reset method be implemented.

C#

```csharp
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}


// Collection class.
public class AllColors : System.Collections.IEnumerable
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
                }
            }

            bool System.Collections.IEnumerator.MoveNext()
            {
                _position++;
                return (_position < _colors.Length);
            }

            void System.Collections.IEnumerator.Reset()
            {
                _position = -1;
            }
        }
    }

    // Element class.
    public class Color
    {
        public string Name { get; set; }
    }
```

Iterators

An *iterator* is used to perform a custom iteration over a collection. An iterator can be a method or a **get** accessor. An iterator uses a yield return statement to return each element of the collection one at a time.

You call an iterator by using a foreach statement. Each iteration of the **foreach** loop calls the iterator. When a **yield return** statement is reached in the iterator, an expression is returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

For more information, see Iterators (C#).

The following example uses an iterator method. The iterator method has a **yield return** statement that is inside a for loop. In the `ListEvenNumbers` method, each iteration of the **foreach** statement body creates a call to the iterator method, which proceeds to the next **yield return** statement.

C#

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}
```

```
    private static IEnumerable<int> EvenSequence(
        int firstNumber, int lastNumber)
    {
        // Yield even numbers in the range.
        for (var number = firstNumber; number <= lastNumber; number++)
        {
            if (number % 2 == 0)
            {
                yield return number;
            }
        }
    }
```

See Also

Object and Collection Initializers (C# Programming Guide)
Programming Concepts (C#)
Option Strict Statement
LINQ to Objects (C#)
Parallel LINQ (PLINQ)
Collections and Data Structures
Creating and Manipulating Collections
Selecting a Collection Class
Comparisons and Sorts Within Collections
When to Use Generic Collections
How to: Access a Collection Class with foreach (C# Programming Guide)