Collections (Visual Basic)

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 in Visual Basic.

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, include Imports statements 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(Of 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 For Each...Next statement.

VB

```
' Create a list of strings.
Dim salmons As New List(Of String)
salmons.Add("chinook")
salmons.Add("coho")
salmons.Add("pink")
salmons.Add("sockeye")

' Iterate through the list.
For Each salmon As String In salmons
    Console.Write(salmon & " ")
Next
'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 Collection Initializers (Visual Basic).

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

VB

```
' Create a list of strings by using a
' collection initializer.
Dim salmons As New List(Of String) From
    {"chinook", "coho", "pink", "sockeye"}

For Each salmon As String In salmons
    Console.Write(salmon & " ")
Next
'Output: chinook coho pink sockeye
```

You can use a For...Next statement instead of a **For Each** 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...Next** instead of **For Each**.

VB

```
Dim salmons As New List(Of String) From
```

```
    {"chinook", "coho", "pink", "sockeye"}

For index = 0 To salmons.Count - 1
    Console.Write(salmons(index) & " ")
Next
'Output: chinook coho pink sockeye
```

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

VB

```
' Create a list of strings by using a
' collection initializer.
Dim salmons As New List(Of String) From
    {"chinook", "coho", "pink", "sockeye"}

' Remove an element in the list by specifying
' the object.
salmons.Remove("coho")

For Each salmon As String In salmons
    Console.Write(salmon & " ")
Next
'Output: chinook pink sockeye
```

The following example removes elements from a generic list. Instead of a **For Each** statement, a For...Next 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.

VB

```
Dim numbers As New List(Of Integer) From
    {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

' Remove odd numbers.
For index As Integer = numbers.Count - 1 To 0 Step -1
    If numbers(index) Mod 2 = 1 Then
        ' Remove the element by specifying
        ' the zero-based index in the list.
        numbers.RemoveAt(index)
    End If
Next

' Iterate through the list.
' A lambda expression is placed in the ForEach method
' of the List(T) object.
numbers.ForEach(
    Sub(number) Console.Write(number & " "))
' Output: 0 2 4 6 8
```

For the type of elements in the List(Of T), you can also define your own class. In the following example, the `Galaxy` class

that is used by the List(Of T) is defined in the code.

VB

```
Private Sub IterateThroughList()
    Dim theGalaxies As New List(Of Galaxy) From
        {
            New Galaxy With {.Name = "Tadpole", .MegaLightYears = 400},
            New Galaxy With {.Name = "Pinwheel", .MegaLightYears = 25},
            New Galaxy With {.Name = "Milky Way", .MegaLightYears = 0},
            New Galaxy With {.Name = "Andromeda", .MegaLightYears = 3}
        }

    For Each theGalaxy In theGalaxies
        With theGalaxy
            Console.WriteLine(.Name & "   " & .MegaLightYears)
        End With
    Next

    ' Output:
    '  Tadpole  400
    '  Pinwheel  25
    '  Milky Way  0
    '  Andromeda  3
End Sub

Public Class Galaxy
    Public Property Name As String
    Public Property MegaLightYears As Integer
End Class
```

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
- Visual Basic **Collection** class

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

Visual Basic Collection Class

You can use the Visual Basic Collection class to access a collection item by using either a numeric index or a **String** key. You can add items to a collection object either with or without specifying a key. If you add an item without a key, you must use its numeric index to access it.

The Visual Basic **Collection** class stores all its elements as type **Object**, so you can add an item of any data type. There is no safeguard against inappropriate data types being added.

When you use the Visual Basic **Collection** class, the first item in a collection has an index of 1. This differs from the .NET Framework collection classes, for which the starting index is 0.

Whenever possible, you should use the generic collections in the System.Collections.Generic namespace or the System.Collections.Concurrent namespace instead of the Visual Basic **Collection** class.

For more information, see Collection.

Implementing a Collection of Key/Value Pairs

The Dictionary(Of 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 **For Each** statement.

VB

```
Private Sub IterateThroughDictionary()
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    For Each kvp As KeyValuePair(Of String, Element) In elements
        Dim theElement As Element = kvp.Value

        Console.WriteLine("key: " & kvp.Key)
```

```
        With theElement
            Console.WriteLine("values: " & .Symbol & " " &
                .Name & " " & .AtomicNumber)
        End With
    Next
End Sub

Private Function BuildDictionary() As Dictionary(Of String, Element)
    Dim elements As New Dictionary(Of String, Element)

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

    Return elements
End Function

Private Sub AddToDictionary(ByVal elements As Dictionary(Of String, Element),
ByVal symbol As String, ByVal name As String, ByVal atomicNumber As Integer)
    Dim theElement As New Element

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

    elements.Add(Key:=theElement.Symbol, value:=theElement)
End Sub

Public Class Element
    Public Property Symbol As String
    Public Property Name As String
    Public Property AtomicNumber As Integer
End Class
```

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

VB

```
Private Function BuildDictionary2() As Dictionary(Of String, Element)
    Return New Dictionary(Of String, Element) From
        {
            {"K", New Element With
                {.Symbol = "K", .Name = "Potassium", .AtomicNumber = 19}},
            {"Ca", New Element With
                {.Symbol = "Ca", .Name = "Calcium", .AtomicNumber = 20}},
            {"Sc", New Element With
                {.Symbol = "Sc", .Name = "Scandium", .AtomicNumber = 21}},
            {"Ti", New Element With
                {.Symbol = "Ti", .Name = "Titanium", .AtomicNumber = 22}}
        }
End Function
```

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)` code in Visual Basic.

VB

```
Private Sub FindInDictionary(ByVal symbol As String)
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    If elements.ContainsKey(symbol) = False Then
        Console.WriteLine(symbol & " not found")
    Else
        Dim theElement = elements(symbol)
        Console.WriteLine("found: " & theElement.Name)
    End If
End Sub
```

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

VB

```
Private Sub FindInDictionary2(ByVal symbol As String)
    Dim elements As Dictionary(Of String, Element) = BuildDictionary()

    Dim theElement As Element = Nothing
    If elements.TryGetValue(symbol, theElement) = False Then
        Console.WriteLine(symbol & " not found")
    Else
        Console.WriteLine("found: " & theElement.Name)
    End If
End Sub
```

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 Visual Basic.

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

VB

```
Private Sub ShowLINQ()
    Dim elements As List(Of Element) = BuildList()

    ' LINQ Query.
    Dim subset = From theElement In elements
                 Where theElement.AtomicNumber < 22
                 Order By theElement.Name

    For Each theElement In subset
        Console.WriteLine(theElement.Name & " " & theElement.AtomicNumber)
    Next

    ' Output:
    '  Calcium 20
    '  Potassium 19
    '  Scandium 21
End Sub

Private Function BuildList() As List(Of Element)
    Return New List(Of Element) From
        {
            {New Element With
                {.Symbol = "K", .Name = "Potassium", .AtomicNumber = 19}},
            {New Element With
                {.Symbol = "Ca", .Name = "Calcium", .AtomicNumber = 20}},
            {New Element With
                {.Symbol = "Sc", .Name = "Scandium", .AtomicNumber = 21}},
            {New Element With
                {.Symbol = "Ti", .Name = "Titanium", .AtomicNumber = 22}}
        }
End Function

Public Class Element
    Public Property Symbol As String
    Public Property Name As String
    Public Property AtomicNumber As Integer
End Class
```

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(Of T). The `Car` class implements the IComparable(Of 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(Of T) causes the **CompareTo** method to be called automatically for the `Car` objects in the **List**.

VB

```
Public Sub ListCars()

    ' Create some new cars.
    Dim cars As New List(Of Car) From
    {
        New Car With {.Name = "car1", .Color = "blue", .Speed = 20},
        New Car With {.Name = "car2", .Color = "red", .Speed = 50},
        New Car With {.Name = "car3", .Color = "green", .Speed = 10},
        New Car With {.Name = "car4", .Color = "blue", .Speed = 50},
        New Car With {.Name = "car5", .Color = "blue", .Speed = 30},
        New Car With {.Name = "car6", .Color = "red", .Speed = 60},
        New Car With {.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.
    For Each thisCar As Car In cars
        Console.Write(thisCar.Color.PadRight(5) & " ")
        Console.Write(thisCar.Speed.ToString & " ")
        Console.Write(thisCar.Name)
        Console.WriteLine()
    Next

    ' Output:
    '  blue  50 car4
    '  blue  30 car5
    '  blue  20 car1
    '  green 50 car7
    '  green 10 car3
    '  red   60 car6
    '  red   50 car2
End Sub

Public Class Car
    Implements IComparable(Of Car)

    Public Property Name As String
    Public Property Speed As Integer
    Public Property Color As String
```

```
    Public Function CompareTo(ByVal other As Car) As Integer _
        Implements System.IComparable(Of Car).CompareTo
        ' 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.
        Dim compare As Integer
        compare = String.Compare(Me.Color, other.Color, True)

        ' If the colors are the same, compare the speeds.
        If compare = 0 Then
            compare = Me.Speed.CompareTo(other.Speed)

            ' Use descending order for speed.
            compare = -compare
        End If

        Return compare
    End Function
End Class
```

Defining a Custom Collection

You can define a collection by implementing the IEnumerable(Of T) or IEnumerable interface. For additional information, see Enumerating a Collection.

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 e76533a9-5033-4a0b-b003-9c2be60d185b#BKMK_KindsOfCollections 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.

VB

```
Public Sub ListColors()
    Dim colors As New AllColors()

    For Each theColor As Color In colors
        Console.Write(theColor.Name & " ")
    Next
    Console.WriteLine()
    ' Output: red blue green
End Sub
```

```
' Collection class.
Public Class AllColors
    Implements System.Collections.IEnumerable

    Private _colors() As Color =
    {
        New Color With {.Name = "red"},
        New Color With {.Name = "blue"},
        New Color With {.Name = "green"}
    }

    Public Function GetEnumerator() As System.Collections.IEnumerator _
        Implements System.Collections.IEnumerable.GetEnumerator

        Return New ColorEnumerator(_colors)

        ' Instead of creating a custom enumerator, you could
        ' use the GetEnumerator of the array.
        'Return _colors.GetEnumerator
    End Function

    ' Custom enumerator.
    Private Class ColorEnumerator
        Implements System.Collections.IEnumerator

        Private _colors() As Color
        Private _position As Integer = -1

        Public Sub New(ByVal colors() As Color)
            _colors = colors
        End Sub

        Public ReadOnly Property Current() As Object _
            Implements System.Collections.IEnumerator.Current
            Get
                Return _colors(_position)
            End Get
        End Property

        Public Function MoveNext() As Boolean _
            Implements System.Collections.IEnumerator.MoveNext
            _position += 1
            Return (_position < _colors.Length)
        End Function

        Public Sub Reset() Implements System.Collections.IEnumerator.Reset
            _position = -1
        End Sub
    End Class
End Class

' Element class.
Public Class Color
```

```
    Public Property Name As String
End Class
```

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 statement to return each element of the collection one at a time.

You call an iterator by using a For Each...Next statement. Each iteration of the **For Each** loop calls the iterator. When a **Yield** 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 (Visual Basic).

The following example uses an iterator method. The iterator method has a **Yield** statement that is inside a For...Next loop. In the `ListEvenNumbers` method, each iteration of the **For Each** statement body creates a call to the iterator method, which proceeds to the next **Yield** statement.

VB

```
Public Sub ListEvenNumbers()
    For Each number As Integer In EvenSequence(5, 18)
        Console.Write(number & " ")
    Next
    Console.WriteLine()
    ' Output: 6 8 10 12 14 16 18
End Sub

Private Iterator Function EvenSequence(
ByVal firstNumber As Integer, ByVal lastNumber As Integer) _
As IEnumerable(Of Integer)

' Yield even numbers in the range.
    For number = firstNumber To lastNumber
        If number Mod 2 = 0 Then
            Yield number
        End If
    Next
End Function
```

See Also

Collection Initializers (Visual Basic)
Programming Concepts (Visual Basic)
Option Strict Statement
LINQ to Objects (Visual Basic)
Parallel LINQ (PLINQ)

Collection Initializers (Visual Basic)

Visual Studio 2015

Collection initializers provide a shortened syntax that enables you to create a collection and populate it with an initial set of values. Collection initializers are useful when you are creating a collection from a set of known values, for example, a list of menu options or categories, an initial set of numeric values, a static list of strings such as day or month names, or geographic locations such as a list of states that is used for validation.

For more information about collections, see Collections (C# and Visual Basic).

You identify a collection initializer by using the **From** keyword followed by braces (**{}**). This is similar to the array literal syntax that is described in Arrays in Visual Basic. The following examples show various ways to use collection initializers to create collections.

VB

```
' Create an array of type String().
Dim winterMonths = {"December", "January", "February"}

' Create an array of type Integer()
Dim numbers = {1, 2, 3, 4, 5}

' Create a list of menu options. (Requires an extension method
' named Add for List(Of MenuOption)
Dim menuOptions = New List(Of MenuOption) From {{1, "Home"},
                                                {2, "Products"},
                                                {3, "News"},
                                                {4, "Contact Us"}}
```

Note

C# also provides collection initializers. C# collection initializers provide the same functionality as Visual Basic collection initializers. For more information about C# collection initializers, see Object and Collection Initializers (C# Programming Guide).

Syntax

A collection initializer consists of a list of comma-separated values that are enclosed in braces (**{}**), preceded by the **From** keyword, as shown in the following code.

VB

```
Dim names As New List(Of String) From {"Christa", "Brian", "Tim"}
```

When you create a collection, such as a List(Of T) or a Dictionary(Of TKey, TValue), you must supply the collection type before the collection initializer, as shown in the following code.

VB

```
Public Class AppMenu
    Public Property Items As List(Of String) =
        New List(Of String) From {"Home", "About", "Contact"}
End Class
```

Note

You cannot combine both a collection initializer and an object initializer to initialize the same collection object. You can use object initializers to initialize objects in a collection initializer.

Creating a Collection by Using a Collection Intializer

When you create a collection by using a collection initializer, each value that is supplied in the collection initializer is passed to the appropriate **Add** method of the collection. For example, if you create a List(Of T) by using a collection initializer, each string value in the collection initializer is passed to the Add method. If you want to create a collection by using a collection initializer, the specified type must be valid collection type. Examples of valid collection types include classes that implement the IEnumerable(Of T) interface or inherit the CollectionBase class. The specified type must also expose an **Add** method that meets the following criteria.

- The **Add** method must be available from the scope in which the collection initializer is being called. The **Add** method does not have to be public if you are using the collection initializer in a scenario where non-public methods of the collection can be accessed.
- The **Add** method must be an instance member or **Shared** member of the collection class, or an extension method.
- An **Add** method must exist that can be matched, based on overload resolution rules, to the types that are supplied in the collection initializer.

For example, the following code example shows how to create a `List(Of Customer)` collection by using a collection initializer. When the code is run, each `Customer` object is passed to the `Add(Customer)` method of the generic list.

VB

```
Dim customers = New List(Of Customer) From
    {
        New Customer("City Power & Light", "http://www.cpandl.com/"),
        New Customer("Wide World Importers", "http://www.wideworldimporters.com/"),
        New Customer("Lucerne Publishing", "http://www.lucernepublishing.com/")
    }
```

The following code example shows equivalent code that does not use a collection initializer.

VB

```
Dim customers = New List(Of Customer)
customers.Add(New Customer("City Power & Light", "http://www.cpandl.com/"))
customers.Add(New Customer("Wide World Importers", "http://www.wideworldimporters.com
/"))
customers.Add(New Customer("Lucerne Publishing", "http://www.lucernepublishing.com/"))
```

If the collection has an **Add** method that has parameters that match the constructor for the `Customer` object, you could nest parameter values for the **Add** method within collection initializers, as discussed in the next section. If the collection does not have such an **Add** method, you can create one as an extension method. For an example of how to create an **Add** method as an extension method for a collection, see How to: Create an Add Extension Method Used by a Collection Initializer (Visual Basic). For an example of how to create a custom collection that can be used with a collection initializer, see How to: Create a Collection Used by a Collection Initializer (Visual Basic).

Nesting Collection Initializers

You can nest values within a collection initializer to identify a specific overload of an **Add** method for the collection that is being created. The values passed to the **Add** method must be separated by commas and enclosed in braces (**{}**), like you would do in an array literal or collection initializer.

When you create a collection by using nested values, each element of the nested value list is passed as an argument to the **Add** method that matches the element types. For example, the following code example creates a Dictionary(Of TKey, TValue) in which the keys are of type **Integer** and the values are of type **String**. Each of the nested value lists is matched to the Add method for the **Dictionary**.

VB

```
Dim days = New Dictionary(Of Integer, String) From
    {{0, "Sunday"}, {1, "Monday"}}
```

The previous code example is equivalent to the following code.

VB

```
Dim days = New Dictionary(Of Integer, String)
days.Add(0, "Sunday")
days.Add(1, "Monday")
```

Only nested value lists from the first level of nesting are sent to the **Add** method for the collection type. Deeper levels of nesting are treated as array literals and the nested value lists are not matched to the **Add** method of any collection.

Related Topics

Title	Description

How to: Create an Add Extension Method Used by a Collection Initializer (Visual Basic)	Shows how to create an extension method called **Add** that can be used to populate a collection with values from a collection initializer.
How to: Create a Collection Used by a Collection Initializer (Visual Basic)	Shows how to enable use of a collection initializer by including an **Add** method in a collection class that implements **IEnumerable**.

See Also

Collections (C# and Visual Basic)
Arrays in Visual Basic
Object Initializers: Named and Anonymous Types (Visual Basic)
New Operator (Visual Basic)
Auto-Implemented Properties (Visual Basic)
How to: Initialize an Array Variable in Visual Basic
Local Type Inference (Visual Basic)
Anonymous Types (Visual Basic)
Introduction to LINQ in Visual Basic
How to: Create a List of Items

How to: Create an Add Extension Method Used by a Collection Initializer (Visual Basic)

Visual Studio 2015

When you use a collection initializer to create a collection, the Visual Basic compiler searches for an **Add** method of the collection type for which the parameters for the **Add** method match the types of the values in the collection initializer. This **Add** method is used to populate the collection with the values from the collection initializer.

If no matching **Add** method exists and you cannot modify the code for the collection, you can add an extension method called **Add** that takes the parameters that are required by the collection initializer. This is typically what you need to do when you use collection initializers for generic collections.

Example

The following example shows how to add an extension method to the generic List(Of T) type so that a collection initializer can be used to add objects of type `Employee`. The extension method enables you to use the shortened collection initializer syntax.

VB

```
Public Class Employee
    Public Property Id() As Integer
    Public Property Name() As String
End Class
```

VB

```
Imports System.Runtime.CompilerServices

Module Module1

    <Extension()>
    Sub Add(ByVal list As List(Of Employee), ByVal id As Integer,
                                             ByVal name As String)
        list.Add(New Employee With {.Id = id, .Name = name})
    End Sub

End Module
```

VB

```
Sub Main()
    Dim employees = New List(Of Employee) From {{1, "Adams, Ellen"},
                                                {2, "Hamilton, James R."},
                                                {3, "Ihrig, Ryan"}}
```

```
    End Sub
```

See Also

Collection Initializers (Visual Basic)
How to: Create a Collection Used by a Collection Initializer (Visual Basic)

How to: Create a Collection Used by a Collection Initializer (Visual Basic)

Visual Studio 2015

When you use a collection initializer to create a collection, the Visual Basic compiler searches for an **Add** method of the collection type for which the parameters for the **Add** method match the types of the values in the collection initializer. This **Add** method is used to populate the collection with the values from the collection initializer.

Example

The following example shows an `OrderCollection` collection that contains a public **Add** method that a collection initializer can use to add objects of type `Order`. The **Add** method enables you to use the shortened collection initializer syntax.

VB

```
Public Class Customer
    Public Property Id As Integer
    Public Property Name As String
    Public Property Orders As OrderCollection

    Public Sub New(ByVal id As Integer, ByVal name As String, ByVal orders As
OrderCollection)
        Me.Id = id
        Me.Name = name
        Me.Orders = orders
    End Sub
End Class

Public Class Order
    Public Property Id As Integer
    Public Property CustomerId As Integer
    Public Property OrderDate As DateTime

    Public Sub New(ByVal id As Integer,
                   ByVal customerId As Integer,
                   ByVal orderDate As DateTime)
        Me.Id = id
        Me.CustomerId = customerId
        Me.OrderDate = orderDate
    End Sub
End Class
```

VB

```
Public Class OrderCollection
    Implements IEnumerable(Of Order)
```

```
    Dim items As New List(Of Order)

    Public Property Item(ByVal index As Integer) As Order
        Get
            Return CType(Me(index), Order)
        End Get
        Set(ByVal value As Order)
            items(index) = value
        End Set
    End Property

    Public Sub Add(ByVal id As Integer, ByVal customerID As Integer, ByVal orderDate As
DateTime)
        items.Add(New Order(id, customerID, orderDate))
    End Sub

    Public Function GetEnumerator() As IEnumerator(Of Order) Implements IEnumerable(Of
Order).GetEnumerator
        Return items.GetEnumerator()
    End Function

    Public Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
        Return Me.GetEnumerator()
    End Function
End Class
```

VB

```
Imports System.Runtime.CompilerServices

Module Module1

    <Extension()>
    Sub Add(ByVal genericList As List(Of Customer),
            ByVal id As Integer,
            ByVal name As String,
            ByVal orders As OrderCollection)

        genericList.Add(New Customer(id, name, orders))
    End Sub
End Module
```

VB

```
Dim customerList = New List(Of Customer) From
  {
    {1, "John Rodman", New OrderCollection From {{9, 1, #6/12/2008#},
                                                 {8, 1, #6/11/2008#},
                                                 {5, 1, #5/1/2008#}}},
    {2, "Ariane Berthier", New OrderCollection From {{2, 2, #1/18/2008#},
                                                     {4, 2, #3/8/2008#},
```

```
                                                    {6, 2, #3/18/2008#},
                                                    {7, 2, #5/14/2008#},
                                                    {5, 2, #4/4/2008#}}},
        {3, "Brian Perry", New OrderCollection From {{1, 3, #1/15/2008#},
                                                     {3, 3, #3/8/2008#}}}
    }
```

See Also

Collection Initializers (Visual Basic)
How to: Create an Add Extension Method Used by a Collection Initializer (Visual Basic)