Chapter 9: Generics, Lambdas and LINQ

# Objectives

* + Generics
  + Lambda Expressions
  + Anonymous Types
  + Language Integrated Query (LINQ)

This chapter will introduce you to some more advanced topics that the previous chapter introduced. We will begin with an examination of generics, which is a way to create open ended data structures like the Dictionary, Stack and Queue but that also allows for type safe checking of the data being “bound” to the container. We will also look at lambda functions which are ways of creating throwaway functions, e.g. you need the function then and there, but nowhere else. Finally, we conclude with looking at LINQ which is a super powerful .NET utility allowing you to talk to any data container and bring back results from it. If you haven’t run across LINQ before, prepare to be amazed.

# Generics

When you think about using the data containers that we looked at in the last chapter, it becomes pretty safe to say that we need to worry about the data types that will be stored in the container. Take the lowly array for example. What happens if we create an array of strings and then try to use the Array.Average function on that array? Ouch! Interesting problem, right?

So, while we want our data containers to be able to hold any type of data that we desire, we also want the containers to be type safe, e.g. we want to make sure that whatever we throw in can be correctly acted upon. This is the idea behind generics. While type safety is our number one concern with generics, we also need to realize that there are performance issues with data structure containers. Why do we favor arrays over linked lists? Well, arrays run faster because they are contiguous in memory whereas pointer/reference based structures are dynamically allocated and who knows where the next item lives – locating it is simply costlier than using a contiguous structure like an array.

## Boxing and Unboxing

Remember that we have two different types of data: value types (like Integer) and reference types (like Strings). The value types are the more simplistic pieces of data which can be stored on a stack, while reference types have pointers attached to them and are stored out on the heap. Occasionally we need to convert one category type to the other, e.g. value to reference or reference to value. This is technically called boxing and unboxing.

Let’s say that we have a simple integer variable and we need to convert it into a reference type, we could write a Boxing routine to do this:

'Chapter 9 – Program 1

Function BoxIntegerToObject(ByVal intVariable As Integer) As Object

'Box it

Dim BoxedIntegerObject As Object = intVariable

Return BoxedIntegerObject

End Function

Likewise, we could write an unboxing routine to convert the reference type back into a primitive value type:

Function UnboxObjectToInteger(ByVal BoxedIntegerObject As Object) As Integer

'Unbox it

Dim intVariable As Integer = DirectCast(BoxedIntegerObject, Integer)

Return intVariable

End Function

So here’s our Sub Main to test our boxing/unboxing out:

Sub Main()

'Create an integer variable

Dim aSimpleVar As Integer = 10

Dim unboxedInt As Integer

Dim boxedInt As Object

boxedInt = BoxIntegerToObject(aSimpleVar)

Debug.WriteLine("Try to add 5 to object version " & (boxedInt.ToString + 5))

unboxedInt = UnboxObjectToInteger(boxedInt)

Debug.WriteLine("Try to add 5 to integer version " & (unboxedInt + 5))

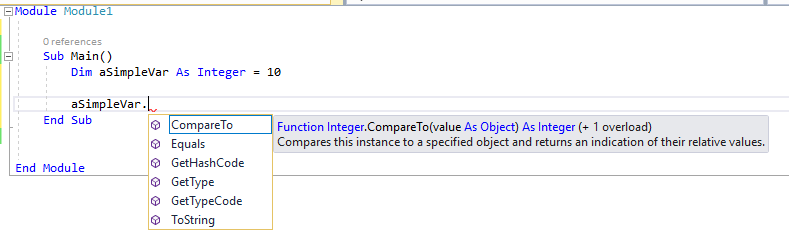
End Sub

Output:

Try to add 5 to object version 15

Try to add 5 to integer version 15

.NET automatically handles a whole bunch of boxing and unboxing for you. Notice that you can throw an integer value variable type in the code editor, press a . after the variable name, and you are provided with a list of methods you can have act upon that integer variable.



But, I thought Integer was a primitive value type? It is – this is .NET autoboxing for you. Some languages like Java actually create two different types: int for primitive value types and then Integer for the reference type. The boxing/unboxing is much more explicit in that language.

.NET on the other hand tries to make the process more transparent – but this comes at a huge penalty. The non-generic containers like ArrayList have most of their methods set up to accept arguments of type Object. So, if you create a ArrayList of integers, guess what’s happening behind the scenes: boxing & unboxing, which is expensive performance-wise. Furthermore, as we start writing code that acts on primitive value data types, we are going to have to ensure that we handle the boxing/unboxing to return the boxed objects back to their correct primitive data types.

## Generic Code before Generics

So what did we do before generics if we wanted to create as type-safe code as possible? We would build up a helper class to do this. Take a look at the following code example.

Let’s say we want to keep information about employees. We’d probably start by building a simple class that holds the information we want, along with a simple constructor and an Employee overridden specific version of the ToString method:

'Chapter 9 - Program 2

Public Class clsEmployee

Public Property strName As String

Public Property intHours As Integer

Public Property sngWage As Single

Public Sub New(ByVal Name As String, ByVal Hours As Integer,

ByVal Wage As Single)

Me.strName = Name

Me.intHours = Hours

Me.sngWage = Wage

End Sub

Public Overrides Function ToString() As String

Return String.Format("Name: {0}, Hours: {1}, Wage: {2}",

strName, intHours, sngWage)

End Function

End Class

I now want to create an ArrayList that can only hold only clsEmployee instances to keep things as type safe as possible. I also want to be able to walk across my ArrayList using the For Each iterator, so I will have to implement the IEnumerable interface. Finally, I’ll add a couple of other helper methods that would be useful to this class:

Public Class ArrayListOfEmployees

'Support the For Each construct

Implements IEnumerable

Private arrEmployees As New ArrayList()

'Only allow Employee instances in here

Public Sub AddEmployee(ByVal AnEmployee As clsEmployee)

arrEmployees.Add(AnEmployee)

End Sub

'Clear out all employees

Public Sub ClearAllEmployees()

arrEmployees.Clear()

End Sub

'Return a count of how many people are in here

Public ReadOnly Property Count() As Integer

Get

Return arrEmployees.Count

End Get

End Property

'Code to handle getting each instance in a For Each construct

Public Function GetEnumerator() As IEnumerator \_

Implements IEnumerable.GetEnumerator

'Simply return the iterator to the underlying ArrayList class

'that we used as the basis for this class

Return arrEmployees.GetEnumerator()

End Function

End Class

Now we’re ready to take the classes for a test drive…

Sub Main()

Dim myEmployees As New ArrayListOfEmployees

myEmployees.AddEmployee(New clsEmployee("Sue", 40, 20.0))

myEmployees.AddEmployee(New clsEmployee("Bill", 38, 18.0))

myEmployees.AddEmployee(New clsEmployee("Tom", 42, 19.0))

myEmployees.AddEmployee(New clsEmployee("Liz", 41, 19.0))

For Each anEmp In myEmployees

Debug.WriteLine(anEmp.ToString())

Next

End Sub

Here’s the output as you’d expect – we created and stored four employee objects:

Name: Sue, Hours: 40, Wage: 20

Name: Bill, Hours: 38, Wage: 18

Name: Tom, Hours: 42, Wage: 19

Name: Liz, Hours: 41, Wage: 19

What you should conclude from the previous code is that it wasn’t all that difficult to construct our own form of a type specific ArrayList that would only allow instances of clsEmployee to be placed in it. Now let’s say that we want to do the same thing for book objects. Here’s the resulting code:

'Chapter 9 - Program 3

Public Class clsBook

Public Property strTitle As String

Public Property strAuthor As String

Public Property intPages As Integer

Public Property sngPrice As Single

Public Sub New(ByVal Title As String, ByVal Author As String,

ByVal Pages As Integer, ByVal Price As Single)

Me.strTitle = Title

Me.strAuthor = Author

Me.intPages = Pages

Me.sngPrice = Price

End Sub

Public Overrides Function ToString() As String

Return String.Format("Title: {0}, Author: {1}, Pages: {2}, Price: {3}",

strTitle, strAuthor, intPages, sngPrice)

End Function

End Class

Public Class ArrayListOfBooks

'Support the For Each construct

Implements IEnumerable

Private arrBooks As New ArrayList()

'Only allow Book instances in here

Public Sub AddBook(ByVal ABook As clsBook)

arrBooks.Add(ABook)

End Sub

'Clear out all books

Public Sub ClearAllBooks()

arrBooks.Clear()

End Sub

'Return a count of how many books are in here

Public ReadOnly Property Count() As Integer

Get

Return arrBooks.Count

End Get

End Property

'Code to handle getting each instance in a For Each construct

Public Function GetEnumerator() As IEnumerator \_

Implements IEnumerable.GetEnumerator

Return arrBooks.GetEnumerator()

End Function

End Class

Sub Main()

Dim myBooks As New ArrayListOfBooks

myBooks.AddBook(New clsBook("Internet for Fun", "T.R. Williams", 154, 27.5))

myBooks.AddBook(New clsBook("VB Rocket Science", "J.J. Davis", 308, 39.95))

myBooks.AddBook(New clsBook("CIS Poetry", "M.M. Bidgoli", 107, 19.95))

For Each aBook In myBooks

Debug.WriteLine(aBook.ToString())

Next

End Sub

Here’s what we see when we run that program:

Title: Internet for Fun, Author: T.R. Williams, Pages: 154, Price: 27.5

Title: VB Rocket Science, Author: J.J. Davis, Pages: 308, Price: 39.95

Title: CIS Poetry, Author: M.M. Bidgoli, Pages: 107, Price: 19.95

Now, if you just sat back and said “Wait a minute! That code is almost identical to the program before…”, you get a golden star because you’re absolutely right! Obviously the clsBook class will vary only by the data that we want to store in it. Employees have three data items: names, hours and wages, whereas books have four data items: titles, authors, pages and prices. It should make sense that the base class data definition of what makes up an employee versus a book will indeed look different! But did you notice that the ArrayListOfEmployees and the ArrayListOfBooks code looked identical save for the fact that one was working on clsEmployee objects while the other was working on clsBook objects? Extrapolating this concept a little further, if we created 10 more base classes to store data, we’d write 10 more ArrayListOf-type classes too. That sounds like way too much work and repetitive code…

## Solving the Redundant Code Problem with Generics

Now it’s time for generics to come to the rescue. You’ll remember that some of the data containers that we looked at in the previous chapter used this (Of T) type syntax – Dictionaries for example. Those were generic containers, e.g. they are type safe and only allow you to add elements of a particular type. Let’s review some code. I still am going to need to create my base data container classes – there is simply no way around that since these types of classes determine what data we intend to work with… We’ll show how easy it is using Generics by writing code to handle *both* the employees and books data types that we looked at above:

'Chapter 9 - Program 4

Public Class clsEmployee

Public Property strName As String

Public Property intHours As Integer

Public Property sngWage As Single

Public Sub New(ByVal Name As String, ByVal Hours As Integer,

ByVal Wage As Single)

Me.strName = Name

Me.intHours = Hours

Me.sngWage = Wage

End Sub

Public Overrides Function ToString() As String

Return String.Format("Name: {0}, Hours: {1}, Wage: {2}",

strName, intHours, sngWage)

End Function

End Class

Public Class clsBook

Public Property strTitle As String

Public Property strAuthor As String

Public Property intPages As Integer

Public Property sngPrice As Single

Public Sub New(ByVal Title As String, ByVal Author As String,

ByVal Pages As Integer, ByVal Price As Single)

Me.strTitle = Title

Me.strAuthor = Author

Me.intPages = Pages

Me.sngPrice = Price

End Sub

Public Overrides Function ToString() As String

Return String.Format("Title: {0}, Author: {1}, Pages: {2}, Price: {3}",

strTitle, strAuthor, intPages, sngPrice)

End Function

End Class

Main is where all of the real changes take place. We will use the generic List container now. List requires that you identify the type of data you will be using when you declare the List itself.

Sub Main()

Dim myEmployees As New List(Of clsEmployee)

Dim myBooks As New List(Of clsBook)

myEmployees.Add(New clsEmployee("Sue", 40, 20.0))

myEmployees.Add(New clsEmployee("Bill", 38, 18.0))

myEmployees.Add(New clsEmployee("Tom", 42, 19.0))

myEmployees.Add(New clsEmployee("Liz", 41, 19.0))

For Each anEmp In myEmployees

Debug.WriteLine(anEmp.ToString())

Next

myBooks.Add(New clsBook("Internet for Fun", "T.R. Williams", 154, 27.5))

myBooks.Add(New clsBook("VB Rocket Science", "J.J. Davis", 308, 39.95))

myBooks.Add(New clsBook("CIS Poetry", "M.M. Bidgoli", 107, 19.95))

For Each aBook In myBooks

Debug.WriteLine(aBook.ToString())

Next

End Sub

Output:

Name: Sue, Hours: 40, Wage: 20

Name: Bill, Hours: 38, Wage: 18

Name: Tom, Hours: 42, Wage: 19

Name: Liz, Hours: 41, Wage: 19

Title: Internet for Fun, Author: T.R. Williams, Pages: 154, Price: 27.5

Title: VB Rocket Science, Author: J.J. Davis, Pages: 308, Price: 39.95

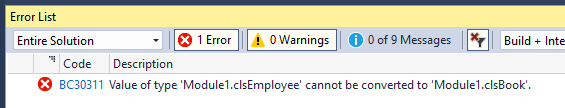
Title: CIS Poetry, Author: M.M. Bidgoli, Pages: 107, Price: 19.95

How was that for easy? Instead of writing the container class wrapper myself to try to ensure type safety like we did above in the first two examples, the generic List class takes care of this for us! Now I can concentrate on writing my base data classes and let the generic classes handle the details.

If you doubt the type safety of List try to enter and compile this line of code:

myBooks.Add(New clsEmployee("Moe", 40, 25))

The compiler gets upset as soon as you add it to the code base, telling you that it can’t convert a clsEmployee instance into a clsBook instance!



Microsoft has optimized these classes to be better performers than their non-generic counterparts. If you create a List(Of Integer), then the data type is Integer in nature (value type) and no hidden boxing/unboxing occurs to get it to/from a reference type. Here’s a list of the generic container classes we’ve looked at:

* Dictionary(Of TKey, TValue)
* List(Of T)
* LinkedList(Of T)
* Queue(Of T)
* SortedDictionary(Of T)
* SortedSet(Of T)
* Stack(Of T)

## Generics Initialization

While we examined various containers last chapter, I would like to add a couple new syntactical savings mechanisms when initializing a generic container:

'Chapter 9 - Program 5

Public Class clsEmployee

Public Property strName As String

Public Property intHours As Integer

Public Property sngWage As Single

'To support With From notation on initialization

Public Sub New()

End Sub

Public Sub New(ByVal Name As String, ByVal Hours As Integer,

ByVal Wage As Single)

Me.strName = Name

Me.intHours = Hours

Me.sngWage = Wage

End Sub

Public Overrides Function ToString() As String

Return String.Format("Name: {0}, Hours: {1}, Wage: {2}",

strName, intHours, sngWage)

End Function

End Class

Sub Main()

'We can initialize collections with an Add method using something

'that looks like array initialization notation

Dim myNumbers As New List(Of Integer) From {2, 4, 6, 8, 10}

'We can also use the From and With notation to handle instances of

'other classes. Note that there was a change in the class itself

'We had to add another constructor that takes no arguments and

'then we can see the public properties -- design-wise this is

'not a good way to go!

Dim myEmployees As New List(Of clsEmployee) From {

New clsEmployee() With {.strName = "Sue", .intHours = 40, .sngWage = 20.0},

New clsEmployee() With {.strName = "Bill", .intHours = 38, .sngWage = 18.0},

New clsEmployee() With {.strName = "Tom", .intHours = 42, .sngWage = 19.0},

New clsEmployee() With {.strName = "Liz", .intHours = 41, .sngWage = 19.0}

}

For Each anEmp In myEmployees

Debug.WriteLine(anEmp.ToString())

Next

End Sub

Here’s the program output:

Name: Sue, Hours: 40, Wage: 20

Name: Bill, Hours: 38, Wage: 18

Name: Tom, Hours: 42, Wage: 19

Name: Liz, Hours: 41, Wage: 19

## Creating Generic Code

Now that we’ve got a pretty good idea of what a generic class is, how do we start writing our own generic methods? What we really want to be able to do is write code once and ship basically any data type through. Think back to 216: what really changed on most data structures? Just the data that was going to be stored. Wouldn’t it have been nice to write methods once that could have worked on any data that was thrown at them?

Let’s consider the simplest of functions that we could write: swapping two values. Remember that swapping will involve three items of the same type, two are the original values and then there must be a temporary placeholder. Consider the following code which will implement a generic swap:

'Chapter 9 - Program 6

Public Class clsBook

Public Property strTitle As String

Public Property strAuthor As String

Public Property intPages As Integer

Public Property sngPrice As Single

Public Sub New(ByVal Title As String, ByVal Author As String,

ByVal Pages As Integer, ByVal Price As Single)

Me.strTitle = Title

Me.strAuthor = Author

Me.intPages = Pages

Me.sngPrice = Price

End Sub

Public Overrides Function ToString() As String

Return String.Format("Title: {0}, Author: {1}, Pages: {2}, Price: {3}",

strTitle, strAuthor, intPages, sngPrice)

End Function

End Class

Here’s the star of the show, Swap(Of T). We can name the T placeholder whatever we want, but most .NET conventions just follow T for Template or Type. Notice that the user will need to specify the data type of items being sent in along with the two items themselves. From there on out the code should be pretty familiar to you:

Sub Swap(Of T)(ByRef item1 As T, ByRef item2 As T)

Dim tempItem As T

tempItem = item1

item1 = item2

item2 = tempItem

End Sub

Let’s execute a program to try out swapping on two completely different types of data, one a reference type and the other a value type:

Sub Main()

Dim bookA As New clsBook("Zoology", "Jones", 9999, 99.99)

Dim bookB As New clsBook("Astronomy", "Smith", 100, 111.11)

Dim int1 As Integer = 111

Dim int2 As Integer = 999

Debug.WriteLine("Original bookA : " & bookA.ToString())

Debug.WriteLine("Original bookB : " & bookB.ToString())

Swap(Of clsBook)(bookA, bookB)

Debug.WriteLine("Swapped bookA : " & bookA.ToString())

Debug.WriteLine("Swapped bookB : " & bookB.ToString())

Debug.WriteLine("Original int1 : " & int1)

Debug.WriteLine("Original int2 : " & int2)

Swap(Of Integer)(int1, int2)

Debug.WriteLine("Swapped int1 : " & int1)

Debug.WriteLine("Swapped int2 : " & int2)

End Sub

Sure enough, the swapping code works:

Original bookA : Title: Zoology, Author: Jones, Pages: 9999, Price: 99.99

Original bookB : Title: Astronomy, Author: Smith, Pages: 100, Price: 111.11

Swapped bookA : Title: Astronomy, Author: Smith, Pages: 100, Price: 111.11

Swapped bookB : Title: Zoology, Author: Jones, Pages: 9999, Price: 99.99

Original int1 : 111

Original int2 : 999

Swapped int1 : 999

Swapped int2 : 111

We could extend this idea on up to creating a complete generic class. Let’s assume that we want to create a generic class that can handle working with rectangle data, but we don’t know if the user will be sending our class integer, double or single numeric data. We’ll write this as a more correctly designed class with private data attributes and accessor methods. In any case, generics will come to the rescue for us again:

'Chapter 9 - Program 7

Public Class clsRectangle(Of T)

Private RectWidth As T

Private RectHeight As T

Public Sub New(ByVal width As T, ByVal height As T)

RectWidth = width

RectHeight = height

End Sub

Public Property Width() As T

Get

Return RectWidth

End Get

Set(value As T)

RectWidth = value

End Set

End Property

Public Property Height() As T

Get

Return RectHeight

End Get

Set(value As T)

RectHeight = value

End Set

End Property

Public Overrides Function ToString() As String

Return String.Format("Width {0}, Height {1}", RectWidth, RectHeight)

End Function

Public Sub ResetRectangle()

RectWidth = Nothing

RectHeight = Nothing

End Sub

End Class

Sub Main()

Dim myIntegerRectangle As clsRectangle(Of Integer) =

New clsRectangle(Of Integer)(5, 7)

Dim mySingleRectangle As clsRectangle(Of Single) =

New clsRectangle(Of Single)(10.1, 6.5)

Debug.WriteLine("Integer " & myIntegerRectangle.ToString)

Debug.WriteLine("Single " & mySingleRectangle.ToString)

End Sub

Here’s the program’s output. We can generically handle any numeric data type sent to us with one code implementation:

Integer Width 5, Height 7

Single Width 10.1, Height 6.5

The first thing that I would like to call attention to is the use of the Nothing keyword in the ResetRectangle method. This is a nice way to reset attributes/variables to their underlying default values: 0 for numeric values and Nothing for reference values.

Secondly, it may have dawned on you to try to put in an Area method, I mean after all we have the two values necessary to do the math – unfortunately .NET doesn’t natively allow this kind of functionality to be carried out. Your method attempt would probably look like this:

Public Function CalculateArea() As T

Return RectHeight \* RectWidth

End Function

The compiler is going to complain about this and say that it doesn’t know how to multiply RectHeight & RectWidth because they are of type T and the multiplication operation hasn’t been defined for that type, which is true. There is some method to this madness – what if the types being passed through to the class were actually structures or classes instead of numeric? How would multiply employee1 times employee2? When you remember that generics allow any data type to roll in, the restriction makes sense!

Having said that, though, where there is a programmer there is a way. What if we help things out a little bit as in this:

Public Function CalculateArea() As T

'Notice the function will return whatever type T is

'Let's guarantee that we can do math by checking both pieces

'that we would be trying to multiply together

If IsNumeric(RectHeight) And IsNumeric(RectWidth) Then

'If we try to directly multiply the two terms together,

'the compiler won't allow it. However if we got this

'far, then we know they are numeric, so let's do a little

'sleight-of-hand and box the numerics into an object.

Return CObj(RectHeight) \* CObj(RectWidth)

'Once they are boxed, we know that the runtime environment

'can unbox them for us and do the math -- we just tricked

'the compiler into doing something it believes it can't!

Else

Return Nothing

'If either product is non-numeric, return Nothing because

'an Area calcuation would be nonsensical.

End If

End Function

We’ve got one more small change to make to our program:

'Modify the output in Sub Main to call the CalculateArea method

Debug.WriteLine("Integer " & myIntegerRectangle.ToString &

" - Area = " & myIntegerRectangle.CalculateArea())

Debug.WriteLine("Single " & mySingleRectangle.ToString &

" - Area = " & mySingleRectangle.CalculateArea())

Voila! When we run the application, we see the area correctly calculated:

Integer Width 5, Height 7 - Area = 35

Single Width 10.1, Height 6.5 - Area = 65.65

## Generic Odds and Ends

We started this chapter out by trying to figure out how to create type safe data containers and found that the resulting code was almost identical in each solution. We then learned that generic containers could solve this issue for us and that only the data that we wanted to store would vary. I’d like to revisit the original problem and look at how we could extend one of those generic containers for our own use. Anytime that we are going to use a generic base type, our derived type must allow the generic type to flow through as a type parameter. Take a look at the following code which reworks the clsEmployee example. In this case I am creating a custom MyList(Of clsEmployee) class that uses a List(Of clsEmployee) under the hood to do most of the work:

'Chapter 9 - Program 9

Public Class clsEmployee

Public Property strName As String

Public Property intHours As Integer

Public Property sngWage As Single

Public Sub New(ByVal Name As String, ByVal Hours As Integer,

ByVal Wage As Single)

Me.strName = Name

Me.intHours = Hours

Me.sngWage = Wage

End Sub

Public Overrides Function ToString() As String

Return String.Format("Name: {0}, Hours: {1}, Wage: {2}",

strName, intHours, sngWage)

End Function

End Class

Public Class MyList(Of T)

'We want our custom class to support the For Each iterator

Implements IEnumerable

'Behind the scenes our class will use a List to hold the data

'items -- why write a whole bunch of routines when List will

'do most of the work for us?

Private listItems As New List(Of T)

'Here's out AddItem routine, just like we had before

Public Sub AddItem(ByVal AnItem As T)

'Our AddItem calls the underlying List's Add method

listItems.Add(AnItem)

End Sub

'Clear out all items

Public Sub ClearAllItems()

'Call the underlying List's Clear method

listItems.Clear()

End Sub

'Return a count of how many items are in here

Public ReadOnly Property Count() As Integer

Get

Return listItems.Count

End Get

End Property

'Code to handle getting each instance in a For Each construct

Public Function GetEnumerator() As IEnumerator \_

Implements IEnumerable.GetEnumerator

'Simply return the iterator to the underlying List class'

'iterator that we used as our backing data structure

Return listItems.GetEnumerator()

End Function

End Class

Sub Main()

'Create our generic MyList of clsEmployee instances

Dim myEmployees As New MyList(Of clsEmployee)

'Call our custom AddItem method and use a For Each

'iterator to walk across the objects

myEmployees.AddItem(New clsEmployee("Sue", 40, 20.0))

myEmployees.AddItem(New clsEmployee("Bill", 38, 18.0))

myEmployees.AddItem(New clsEmployee("Tom", 42, 19.0))

myEmployees.AddItem(New clsEmployee("Liz", 41, 19.0))

For Each anEmp In myEmployees

Debug.WriteLine(anEmp.ToString())

Next

End Sub

Program output:

Name: Sue, Hours: 40, Wage: 20

Name: Bill, Hours: 38, Wage: 18

Name: Tom, Hours: 42, Wage: 19

Name: Liz, Hours: 41, Wage: 19

Notice that our code added a (Of T) parameter to the MyList class since it has an underlying List class within it and we have to allow the generic type to flow through.

To a degree, we can constrain what types of entities are allowed into a generic. When we perform the initial declaration of a data type that allows generic through, we use the (Of T) construct. This declaration can be further extended to support the following types:

|  |  |
| --- | --- |
| *Type* | *Meaning* |
| (Of T As Structure) | The type parameter must be a structure |
| (Of T As Class) | The type parameter must be a reference type |
| (Of T As New) | The type parameter must have a default constructor |
| (Of T as *BaseClassName*) | The type parameter must be derived from the class specified |
| (Of T As *InterfaceName*) | The type parameter must implement the interface specified. Multiple interfaces can be listed in a comma-delimited list |

We can specify multiple types in a single declaration:

Public Class MyGeneric1(Of T As {Class, IEnumerable, New})

'Requires a class with a constructor that implements IEnumerable

End Class

Here’s an example of a structure being allowed through to the generic, but not classes:

Sub Swap(Of T As Structure)(ByRef item1 As T, ByRef item2 As T)

'Will swap structures but not classes

End Sub

We can even go as far as allowing multiple generic types through, e.g. every generic parameter does not have to be of the identical type:

Public Class MyGeneric2(Of T1, T2)

'T1 and T2 can be different types if we so choose

Private Member1 As T1

Private Member2 As T2

End Class

# Lambda Expressions

Lambda expressions (or functions) take their name from Lambda Calculus, which forms the foundation of another style of programming called functional programming (think F#). In Lambda Calculus, a Lambda is a function which takes another function as an argument and returns a function as a result. Where did this nuttiness come from? Alonzo Church in the 1930s: basically, he theorized that every computing task could be expressed as a function and that functions could be passed around whenever and however the programmer needed them.

In programming today, Lambda functions are essentially thought of as throwaway functions – you declare them inline as you need them and they are not built for reusability as other subprograms are. Lambda functions are called anonymous functions in other programming languages. Lambdas let you define an object that contains an entire function and they have actually been in programming languages for a long, long time – DEF back in old school BASIC was an example. Here’s the world’s simplest program using a lambda expression:

Sub Main()

Dim IsEven As Func(Of Integer, Boolean) = Function(num As Integer) (num Mod 2) = 0

For intLoop = 1 To 10

Debug.WriteLine("intLoop Value = " & intLoop & " " & " Even? = " &

IsEven(intLoop))

Next

End Sub

Output:

intLoop Value = 1 Even? = False

intLoop Value = 2 Even? = True

intLoop Value = 3 Even? = False

intLoop Value = 4 Even? = True

intLoop Value = 5 Even? = False

intLoop Value = 6 Even? = True

intLoop Value = 7 Even? = False

intLoop Value = 8 Even? = True

intLoop Value = 9 Even? = False

intLoop Value = 10 Even? = True

The first line of Sub Main contains the Lambda expression and that is the Function(num As Integer) (num Mod 2) = 0 line. Notice that this sort of looks like a normal function except that there is no name for the function. It simply says there is an incoming value of type Integer and then you tell what work you want done with that Integer variable, e.g. the function’s purpose. In this case we are asking if there is no remainder when the value in the Integer variable is divided by 2.

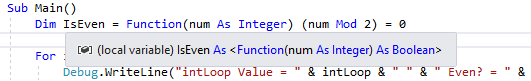
You’ll notice that there is no return statement either. Lambdas don’t specify one – we know that we are comparing the number’s remainder and seeing if it is equal to 0 or not. The result of the comparison will be a True or False value. We are therefore really returning a Boolean value behind the scenes.

Since we’re sending something back (a Boolean value), we need some sort of placeholder to hold both the definition, parameter and the return value. That’s where the first half of the syntax comes in: Dim IsEven As Func(Of Integer, Boolean) = line . This says that we’re declaring a Function variable. Since the word Function was already used to indicate the start of the Function subprogram type (and now Lambda functions when no name is given), we needed a different keyword -- Func was chosen. From the previous discussion on generics, you can see that we are being data type safe in our function variable by requiring the first argument to be Of Integer. This is the value that we pass into the Lambda function. Boolean indicates the type of data coming back from the Lambda function. So, there you have it, a IsEven is a function variable that holds a function in it.

So far, we really haven’t seen anything earthshaking that we couldn’t have written in a non-Lambda form. Thanks to Visual Basic’s ability to infer types, we can create much smaller and tighter Lambda expressions. Consider the following:

Dim IsEven = Function(num As Integer) (num Mod 2) = 0

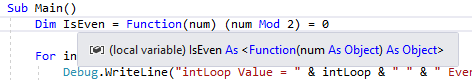
Notice that I didn’t tell VB that IsEven is a function variable (no Func syntax). Nor did I tell what the return type should be from the Lambda. Take a look, though, at what the compiler says when I hover over the IsEven variable:



How’s that for cool? This is type inference coming into play again. We let VB try to determine the type based on what we’re doing and, while in general, type inference may not be a good thing, with Lambdas it’s awesome! Did it guess wrong? No…why? Because it was able to look at the Lambda function itself and tell that from the header that an Integer value would be passed in and based on the return value, a Boolean would be coming out. We formally expressed this in the previous version, but we really didn’t need to thanks to VB’s inference. Going one step further we can do this:

Dim IsEven = Function(num) (num Mod 2) = 0

Notice that I didn’t declare what type num is supposed to be. So now VB will try to infer the type. Nothing other than Integer or Long really makes sense to use here since we would need to cast any real numbers to an Integer before using Mod anyway. Behind the scenes, though, VB can’t infer down to the base type anymore, so it default to Object (because everything can go in an object):



The point is that we’ve written a pretty generic Lambda function now. If we, the programmers, know our intent, then the Lambda function should work just fine for non-floating point numerics.

Is it possible though, to write a Lambda in such a way that any numeric value, whole or floating-point, could be successfully passed in? The answer to that question is “Yes.” Here’s an example of a Lambda where any type of numeric value should be allowed through:

Sub Main()

'We can technically add Integer, Long, Decimal, Single, Double -- if it's

'sent two numerics, this Lambda will work just fine

Dim AddEm = Function(num1, num2) num1 + num2

For intLoop1 = 1 To 3

For intLoop2 = 1 To 3

Debug.WriteLine(intLoop1 & " + " & intLoop2 & " = " & AddEm(intLoop1,

intLoop2))

Next

Next

End Sub

Output:

1 + 1 = 2

1 + 2 = 3

1 + 3 = 4

2 + 1 = 3

2 + 2 = 4

2 + 3 = 5

3 + 1 = 4

3 + 2 = 5

3 + 3 = 6

I used integer values in this code example, but there’s no reason that you couldn’t send a Single, Double or mixed numeric type value set instead. Just remember that if you set Option Strict to On, the type inference won’t work until you add the necessary As pieces. More type safety, but also more code you have to type and more restrictions on what kinds of data that code can work on.

## More Complex Lambda Examples

Lambdas can be designed so that they call functions defined outside of their scope, giving you the ability to build up some pretty complex calculations. Take a look at this example:

Function SquareIt(ByVal num As Integer) As Integer

Return num \* num

End Function

Sub Main()

Dim SquareAndDouble = Function(num) SquareIt(num) \* 2

For intLoop = 1 To 5

Debug.WriteLine("value = " & intLoop & " SquareAndDoubled = " &

SquareAndDouble(intLoop))

Next

End Sub

Output:

value = 1 SquareAndDoubled = 2

value = 2 SquareAndDoubled = 8

value = 3 SquareAndDoubled = 18

value = 4 SquareAndDoubled = 32

value = 5 SquareAndDoubled = 50

Notice that the Lambda is calling an external function, SquareIt and then multiplying the returned result by 2. If you liked that, then you’ll love this: we can even have Lambdas composed of Lambdas (recursive fun, anyone?)

Sub Main()

Dim AddEm = Function(x) Function(y) x + y

For intLoop1 = 1 To 3

For intLoop2 = 1 To 3

Debug.WriteLine(intLoop1 & " + " & intLoop2 & " = " &

AddEm(intLoop1)(intLoop2))

Next

Next

End Sub

This Lambda expression call still requires two parameters to be passed to it. However, they are not comma separated as in the previous AddEm example. There are actually two lambdas that reduce down to one Lambda and then the single Lambda resolves to a result. So if we call AddEm(2)(3) the two Lambda function reduces down to Function(y) 2 + y, which then reduces to 2 + 3, which results in a 5.

If you don’t like thinking about things this way, that’s okay! Remember that we could have used a single Lambda with multiple parameters and avoided this issue. On the con side though, Lambda Calculus was in part developed to see how complex functions could be broken down into more basic functions and that is partially how you must think if you are going to use a functional programming language!

## A Practical Use for Lambdas

The examples we’ve looked at so far with Lambda expressions haven’t been world changers, but instead they were designed to try to get you ready to see where .NET actually lets you use them. Lambdas (or again, anonymous functions/methods) are used quite extensively in the framework. Yes, they are complex and difficult to understand at first, but they can save you a lot of time and coding. Let’s say that we have a List() object that we’ve built up and we want to create a sublist from it that contains all of the even numbers. There is a FindAll method that the List object possess and guess what the argument to FindAll is? Yep -- a Lambda expression that tells what you are interested in finding! Now, how’s that for slick? Here’s the code:

Sub Main()

'Build up the Original List of all numbers

Dim myList As New List(Of Integer) From {18, 23, 27, 11, 4, 16, 48, 53}

'Build up a placeholder for the evens

Dim EvenList As List(Of Integer)

'Provide a lambda to the .FindAll method to get just the even numbers back

EvenList = myList.FindAll(Function(num) (num Mod 2) = 0)

'Here are the even numbers from the original list (stored in EvenList)

For Each aNumber In EvenList

Debug.WriteLine(aNumber)

Next

End Sub

Here’s the output:

18

4

16

48

The example above works fine if Option Strict is set to Off. Again, if I had turned it on, I would need to modify the Lambda line to satisfy the type safety check on the generic function:

EvenList = myList.FindAll(Function(num Of Integer) (num Mod 2) = 0)

Now do you get the power of Lambdas? Visual Basic calls the argument to the FindAll method a predicate. We can use Lambdas (among other things) wherever a predicate shows up, which is in many of our data structures including Arrays and the generic structures like List!

## Some Lambda Odd and Ends

While the examples we’ve looked at so far have been functions, they do not strictly have to be – we can also make Lambda subroutines:

Sub Main()

Dim Greet = Sub(x As String) Debug.WriteLine("Hello " & x)

Greet("Scott")

End Sub

Lambda expressions also do not have to be assigned to variables. We already saw this take place in the previous List even number example’s .FindAll method as a predicate. Take a look at the following example:

Sub Main()

Debug.WriteLine((Function(num As Integer) num + 1)(10))

End Sub

That example places a Lambda expression inline to a WriteLine statement. Lambda expressions (both Sub and Function can be more than a single line long – if they are, then we need to use Return/End Function or End Sub syntax). Let’s rewrite the even number problem as a multiline function:

Sub Main()

'Build up the Original List of all numbers

Dim myList As New List(Of Integer) From {18, 23, 27, 11, 4, 16, 48, 53}

'Build up a placeholder for the evens

Dim EvenList As List(Of Integer)

'Provide a lambda to the .FindAll method to get just the even numbers back

EvenList = myList.FindAll(

Function(num As Integer)

Dim blnResult As Boolean

blnResult = (num Mod 2) = 0

Return blnResult

End Function

)

'Here are the even numbers from the original list (stored in EvenList)

For Each aNumber In EvenList

Debug.WriteLine(aNumber)

Next

End Sub

It’s still a Lambda, just a multiline version. Here are the official rules on Lambdas from the MSDN help:

* A Lambda expression does not have a name.
* Lambda expressions cannot have modifiers, such as Overloads or Overrides.
* Single-line Lambda functions do not use an As clause to designate the return type. Instead, the type is inferred from the value that the body of the Lambda expression evaluates to. For example, if the body of the lambda expression is cust.City = "London", its return type is Boolean.
* In multi-line Lambda functions, you can either specify a return type by using an As clause, or omit the As clause so that the return type is inferred. When the As clause is omitted for a multi-line lambda function, the return type is inferred to be the dominant type from all the Return statements in the multi-line Lambda function. The dominant type is a unique type that all other types supplied to the Return statement can widen to. If this unique type cannot be determined, the dominant type is the unique type that all other types supplied to the Return statement can narrow to. If neither of these unique types can be determined, the dominant type is Object. For example, if the expressions supplied to the Return statement contain values of type Integer, Long, and Double, the resulting type is Double. Both Integer and Long widen to Double and only Double. Therefore, Double is the dominant type.
* The body of a single-line function must be an expression that returns a value, not a statement. There is no Return statement for single-line functions. The value returned by the single-line function is the value of the expression in the body of the function.
* The body of a single-line subroutine must be single-line statement.
* Single-line functions and subroutines do not include an End Function or End Sub statement.
* You can specify the data type of a Lambda expression parameter by using the As keyword, or the data type of the parameter can be inferred. Either all parameters must have specified data types or all must be inferred.
* Optional and ParamArray parameters are not permitted.
* Generic parameters are not permitted.

# Anonymous Types

In the previous section we saw that would could create anonymous methods – Lambda expressions. In other words, the method/function/sub was used where it was needed and then discarded. If we can do that when executing code, can we do something similar to data? The answer is yes and we call these entities anonymous types.

Rather than declaring a formal class, I can simply start describing some data to Visual Basic and it will infer a data type for me. While the data doesn’t have a formal class known to you, VB will generate one behind the scenes to make things work correctly. Take a look at the following code:

Sub Main()

'Notice no formal class in use here -- we just say stamp out an object

'for me with these data characteristics...

Dim Person1 = New With {.Name = "Bill", .Age = 20, .Salary = 45000}

Dim Person2 = New With {.Name = "Sue", .Age = 35, .Salary = 55000}

Dim Person3 = New With {.Name = "Dave", .Age = 24, .Salary = 37000}

Dim Person4 = New With {.Name = "Sara", .Age = 22, .Salary = 40000}

Dim Person5 = New With {.Name = "Lee", .Age = 45, .Salary = 37000}

'Throw them in an array

Dim People(4) As Object

People(0) = Person1

People(1) = Person2

People(2) = Person3

People(3) = Person4

People(4) = Person5

'Iterate across the array accessing each member by name

For Each aPerson In People

Debug.WriteLine("Name {0} - Age {1} - Salary {2}", aPerson.name, aPerson.age,

aPerson.salary)

Next

End Sub

Output from the program:

Name Bill - Age 20 - Salary 45000

Name Sue - Age 35 - Salary 55000

Name Dave - Age 24 - Salary 37000

Name Sara - Age 22 - Salary 40000

Name Lee - Age 45 - Salary 37000

Notice the syntax – I just said I wanted a New (whatever), technically it is a class so it was derived from Object and then VB started inferring the pieces that need to go inside of that class. Name is a string, Age is an integer and Salary is a single. Now depending on what we’re doing, we don’t even have to provide all of the same pieces to each anonymous type:

Sub Main()

Dim Person1 = New With {.Name = "Bill", .Age = 20, .Salary = 45000}

Dim Person2 = New With {.Name = "Sue", .Age = 35, .Salary = 55000}

Dim Person3 = New With {.Name = "Dave", .Age = 24, .Salary = 37000}

'The next two don't have all the same pieces as the first three or

'even each other!

Dim Person4 = New With {.Name = "Sara", .Salary = 40000}

Dim Person5 = New With {.Name = "Lee", .Age = 45}

'Throw them in an array

Dim People(4) As Object

People(0) = Person1

People(1) = Person2

People(2) = Person3

People(3) = Person4

People(4) = Person5

'Iterate across the array accessing each member

For Each aPerson In People

'We can't blindly try to print all data fields because there is

'no guarantee each person will have them -- so get ready for

'possible exceptions.

Try

Debug.WriteLine("Name {0} - Age {1} - Salary {2}", aPerson.name,

aPerson.age, aPerson.salary)

Catch

Debug.WriteLine("Name {0} is either missing age/salary or both!",

aPerson.name)

End Try

Next

End Sub

Output:

Name Bill - Age 20 - Salary 45000

Name Sue - Age 35 - Salary 55000

Name Dave - Age 24 - Salary 37000

Name Sara is either missing age/salary or both!

Name Lee is either missing age/salary or both!

In the last example, we didn’t supply all of the same fields to each anonymous type we created. We were still able to add them all to the array of objects since they are all objects, but we would throw an exception if we tried to “talk” to an attribute that wasn’t there, so we caught those using a Try-Catch block.

One other thing to note is that VB would have internally generated three different classes for you to support your data needs. The first three instances would all have been of the firstclass type created. Since the fourth instance is missing the Age attribute, a second class type would need to be created. Likewise, the fifth instance is missing the Salary attribute. Neither of the two classes generated thus far can accommodate that particular data type, so a third class is generated as a result. We don’t have to worry about any of this because VB is handling it on our behalf behind the scenes.

# LINQ (Language Integrated Query)

While generics and Lambda expressions are pretty powerful critters on their own, they were partially added to the language to support the LINQ programming facilities. We’ve seen that certain data containers like Arrays have the ability to support some aggregate calls like .Average, .Min, .Max, .Count and so forth, but they are still limited. Microsoft decided that it wanted to enable programmers to be able to talk to and reference data over essentially any data structure that exists, not just Arrays. That’s where the idea of LINQ comes from.

The easiest way to think of LINQ is like SQL for everything, not just databases. Probably the most maddening part of learning LINQ if you already know SQL is that the syntax is just enough different to be a real pain. Since you’ve already seen Lambda expressions though, what we’re capable of doing with LINQ shouldn’t surprise you, but the scope of what can be done will! The real plus side to LINQ is that once you’ve learned its syntax you can use it just about everywhere in .NET: Arrays, Collections, Lists, Queues – you name it!

We just looked at the idea of an anonymous type – again LINQ uses these – it’s how data will be returned from your LINQ query into a data container! That data container will be shaped exactly to accommodate whatever data needs to be returned to you and you won’t have to worry about the class name behind the container. Let’s take a look at a simple example of getting the aggregate functions from an Array back using LINQ instead of the method versions.

Sub Main()

Dim MyInts As Integer() = {22, 45, 17, 6, 12, 93, 15, 104}

Dim theMax = Aggregate nums In MyInts Into Max()

Dim theMin = Aggregate nums In MyInts Into Min()

Dim theAverage = Aggregate nums In MyInts Into Average()

Dim theCount = Aggregate nums In MyInts Into Count()

Dim allNumbers As Object

Debug.WriteLine("Statistics " & vbCrLf &

"Max = " & theMax & vbCrLf &

"Min = " & theMin & vbCrLf &

"Average = " & theAverage & vbCrLf &

"Count = " & theCount & vbCrLf &

"Sum = " & Aggregate nums In MyInts Into Sum())

allNumbers = From nums In MyInts Select nums

Debug.Write("Numbers that are in MyInts : ")

For Each aNum In allNumbers

Debug.Write(aNum & " ")

Next

Debug.WriteLine("")

allNumbers = From nums In MyInts Where nums >= 10 And nums <= 99 Select nums

Debug.Write("Two digit numbers that are in MyInts : ")

For Each aNum In allNumbers

Debug.Write(aNum & " ")

Next

Debug.WriteLine("")

End Sub

We start out by creating a simple array of integer numbers. We then create variables with no type specified, so they will be inferred by what’s returned from the LINQ statement to handle most aggregate functions. I put my LINQ statements on single lines, but just like SQL they can span multiple lines.

I then called the Aggregate function to create a Max, Min, Average and Count value for the contents of the array. Notice the syntax Aggregate nums In MyInts: this tells LINQ what to do and where to look. Aggregrate tells it that we want to perform some operation across the contents of the entire array and that a single value will be returned from the aggregate function. nums is just a placeholder variable for the values that are in the container, myInts. We then tell LINQ what type of aggregration we are interested in and, voila, instant solution.

I then wrote those statistical values out in a Debug.WriteLine statement and you’ll notice that I performed the Sum aggregation in the statement itself. You don’t have to necessarily assign the results of a LINQ query to a variable, especially in this case since a single value is returned – we saw this idea with Lambdas.

I then introduced the From clause which tells LINQ it may be returning back 0 or more values from the rest of the LINQ statement. In the first example, I simply returned all numbers that are in the array and then iterated across them to print them out. That’s an important key LINQ piece – the underlying data container must support enumeration or LINQ cannot be used. The good news for you is that almost every VB data container has this interface implemented!

Finally, I asked LINQ just to return the two digit numbers (10 – 99) that exist in the array. Now, the preceding statements might not have been that exciting to you, but that one should be. Imagine having to write the code to do the same thing. Here’s the output of the program:

Statistics

Max = 104

Min = 6

Average = 39.25

Count = 8

Sum = 314

Numbers that are in MyInts : 22 45 17 6 12 93 15 104

Two digit numbers that are in MyInts : 22 45 17 12 93 15

Let’s do a couple more things before moving on to a different example. Let’s say I want things sorted…

Sub Main()

Dim MyInts As Integer() = {22, 45, 17, 6, 12, 93, 15, 104}

Dim allnumbers As Object

allNumbers = From nums In MyInts

Order By nums Descending

Select nums

Debug.Write("Numbers that are in MyInts : ")

For Each aNum In allNumbers

Debug.Write(aNum & " ")

Next

Debug.WriteLine("")

allNumbers = From nums In MyInts

Where nums >= 10 And nums <= 99

Order By nums

Select nums

Debug.Write("Two digit numbers that are in MyInts : ")

For Each aNum In allNumbers

Debug.Write(aNum & " ")

Next

Debug.WriteLine("")

End Sub

Output:

Numbers that are in MyInts : 104 93 45 22 17 15 12 6

Two digit numbers that are in MyInts : 12 15 17 22 45 93

Pretty fancy, eh? And, I didn’t really have to write a lick of “normal” procedural code to pull this off. Try that in Java! There ain’t no LINQ there… Now do realize that I could have used some of the methods like FindAll with a Lambda predicate to do the same thing, but why? I’d rather concentrate on learning a syntactical tool that I can use everywhere and that will allow me to carry out far more complex operations in a much simpler syntactic manner!

While the last example was pretty cool, let’s take a look at a different problem. Let’s say I have some employee objects in a List (hmmm…where have we seen those before?) and want to do some querying there. LINQ can come to our rescue once again:

'Let's use a class

Public Class clsEmployee

Public Name As String

Public Age As Integer

Public Salary As Single

Public Sub New(ByVal Name As String, ByVal Age As Integer, ByVal Salary As

Single)

Me.Name = Name

Me.Age = Age

Me.Salary = Salary

End Sub

End Class

Sub Main()

Dim myEmps As New List(Of clsEmployee)

Dim LINQResults As Object

myEmps.Add(New clsEmployee("Bill", 20, 45000))

myEmps.Add(New clsEmployee("Sue", 35, 55000))

myEmps.Add(New clsEmployee("Dave", 24, 37000))

myEmps.Add(New clsEmployee("Sara", 22, 40000))

myEmps.Add(New clsEmployee("Lee", 45, 37000))

'Who makes between 30,000 and 40,000?

LINQResults = From emps In myEmps

Where emps.Salary >= 30000 And emps.Salary < 40000

Select emps

Debug.Write("Who makes between 30K and 40K? ")

For Each emp In LINQResults

Debug.Write(emp.name & " " & emp.salary & ", ")

Next

Debug.WriteLine("")

'Whose name starts with an S?

LINQResults = From emps In myEmps

Where emps.Name Like "S\*"

Select emps

Debug.Write("Whose name starts with an S? ")

For Each emp In LINQResults

Debug.Write(emp.name & ", ")

Next

Debug.WriteLine("")

End Sub

Here’s the output:

Who makes between 30K and 40K? Dave 37000, Lee 37000,

Whose name starts with an S? Sue, Sara,

That totally rocks! Again, try it in some other language – think of a collection of objects and then that you want to do some querying. You’d have to write an iterator to walk over the objects, write a comparator function to hold the values that match the criteria you’re looking for, et cetera. Way too much work! And, if I decide to change storage containers say from the List to a LinkedList and add a field in the base data class, there is no code changes to the querying portion! Don’t believe me? Just watch:

'Let's use a class

Public Class clsEmployee

Public Name As String

Public Age As Integer

Public Salary As Single

'Add a JobTitle to the existing class

Public JobTitle As String

Public Sub New(ByVal Name As String, ByVal Age As Integer, ByVal Salary As

Single, ByVal JobTitle As String)

Me.Name = Name

Me.Age = Age

Me.Salary = Salary

'Fix the Constructor...

Me.JobTitle = JobTitle

End Sub

End Class

Sub Main()

'We're using a LinkedList now!

Dim myEmps As New LinkedList(Of clsEmployee)

Dim LINQResults As Object

'Change the List.Add method to LinkedList.AddLast

myEmps.AddLast(New clsEmployee("Bill", 20, 45000, "Technician"))

myEmps.AddLast(New clsEmployee("Sue", 35, 55000, "Engineer"))

myEmps.AddLast(New clsEmployee("Dave", 24, 37000, "CIO"))

myEmps.AddLast(New clsEmployee("Sara", 22, 40000, "CEO"))

myEmps.AddLast(New clsEmployee("Lee", 45, 37000, "Scientist"))

'Who makes between 30,000 and 40,000?

LINQResults = From emps In myEmps

Where emps.Salary >= 30000 And emps.Salary < 40000

Select emps

Debug.Write("Who makes between 30K and 40K? ")

For Each emp In LINQResults

Debug.Write(emp.name & " " & emp.salary & ", ")

Next

Debug.WriteLine("")

'Whose name starts with an S?

LINQResults = From emps In myEmps

Where emps.Name Like "S\*"

Select emps

Debug.Write("Whose name starts with an S? ")

For Each emp In LINQResults

Debug.Write(emp.name & ", ")

Next

Debug.WriteLine("")

End Sub

End Module

Same output as before even though we added a new field to clsEmployee and changed the entire data structure from a List to a LinkedList:

Who makes between 30K and 40K? Dave 37000, Lee 37000,

Who's name starts with an S? Sue, Sara,

## LINQ Statements

Well we should probably turn to what we can say with LINQ so that you have some idea of the expressive capability of the language. If you’ve seen SQL before, you’ll get most of this. I’ll throw in an example or two as needed.

|  |  |  |
| --- | --- | --- |
| *Statement* | *Purpose* | *Example* |
| FROM | Specifies the data source to query on | From emps In AllEmployees |
| SELECT | Determines which fields to return | From emps In AllEmployees Select emps.name, emps.age |
| DISTINCT | Returns only unique instances (no duplicates) | From emps In AllEmployees Select emps.age Distinct |
| WHERE | Specifies criteria that must be matched by the instances that are returned | From emps In AllEmployees Where Age >= 21 Select emps.name |
| ORDER BY | Specifies the fields and order that we want sorting applied to (can specify Descending) | From emps in AllEmployees Order By emps.lastname, emps.firstname |
| GROUP BY | Allows us to combine our results set into groupings | From emps in AllEmployees  Group By EmployeeAge = emps.age Into AgedEmployees = Group, Count()  Order By EmployeeAge |
| AGGREGATE | Returns an aggregate value (All, Any, Average, Count, LongCount, Max, Min, Sum) | Aggregate emps In AllEmployees Into Sum(emps.salary) |
| SKIP/SKIP WHILE | Skips over x items in the results (be careful where you place this clause, e.g. before or after the Order By clause) | From emps In AllEmployees Select emps.name Skip 2 |
| TAKE/TAKE WHILE | Keeps only the first x items in the results (again, be careful where you place this clause!) | From emps In AllEmployees Select emps.name Take 2 |

'Let's use a class

Public Class clsEmployee

Public Name As String

Public Age As Integer

Public Salary As Single

Public JobTitle As String

Public Sub New(ByVal Name As String, ByVal Age As Integer, ByVal Salary As

Single, ByVal JobTitle As String)

Me.Name = Name

Me.Age = Age

Me.Salary = Salary

Me.JobTitle = JobTitle

End Sub

End Class

Sub Main()

Dim myEmps As New LinkedList(Of clsEmployee)

Dim LINQResults As Object

myEmps.AddLast(New clsEmployee("Bill", 20, 45000, "Technician"))

myEmps.AddLast(New clsEmployee("Sue", 35, 55000, "Engineer"))

myEmps.AddLast(New clsEmployee("Dave", 24, 37000, "CIO"))

myEmps.AddLast(New clsEmployee("Sara", 22, 40000, "CEO"))

myEmps.AddLast(New clsEmployee("Lee", 45, 37000, "Scientist"))

'Who makes between 30,000 and 40,000 -- just names though...

LINQResults = From emps In myEmps

Where emps.Salary >= 30000 And emps.Salary < 40000

Select emps.Name

Debug.Write("Who makes between 30K and 40K? ")

For Each emp In LINQResults

'No longer an employee -- just a string! emp.Name errors out...

Debug.Write(emp & ", ")

Next

Debug.WriteLine("")

'Whose name starts with an S and has an a somewhere in it, printed backwards?

LINQResults = From emps In myEmps

Where emps.Name Like "S\*" And InStr(emps.Name, "a")

Select StrReverse(emps.Name)

Debug.Write("Whose name starts with an S and has an a in it (backwards)? ")

For Each emp In LINQResults

'No longer an employee -- just a string! emp.Name errors out...

Debug.Write(emp & ", ")

Next

Debug.WriteLine("")

End Sub

Here’s the program output:

Who makes between 30K and 40K? Dave, Lee,

Whose name starts with an S and has an a in it (backwards)? araS,

## Joins

Well if LINQ works great on a single data source, how it would behave on multiple sources? The Join clause allows us to combine various data sources together and returns a result set back to us.

'Chapter 9 - Program 10

Public Class clsEmployee

Public Name As String

Public Age As Integer

Public Salary As Single

Public JobID As Integer

Public Sub New(ByVal Name As String, ByVal Age As Integer,

ByVal Salary As Single, ByVal JobID As Integer)

Me.Name = Name

Me.Age = Age

Me.Salary = Salary

Me.JobID = JobID

End Sub

End Class

Public Class clsJobAssignment

Public JobID As Integer

Public JobTitle As String

Public PayGrade As String

Public Sub New(ByVal JobID As Integer, ByVal JobTitle As String,

ByVal PayGrade As String)

Me.JobID = JobID

Me.JobTitle = JobTitle

Me.PayGrade = PayGrade

End Sub

End Class

Sub Main()

'We're using a LinkedList now!

Dim myEmps As New LinkedList(Of clsEmployee)

Dim myJobs As New LinkedList(Of clsJobAssignment)

Dim LINQResults As Object

'All employees

myEmps.AddLast(New clsEmployee("Bill", 20, 45000, 415))

myEmps.AddLast(New clsEmployee("Sue", 35, 55000, 305))

myEmps.AddLast(New clsEmployee("Dave", 24, 37000, 200))

myEmps.AddLast(New clsEmployee("Sara", 22, 40000, 104))

'Note this person's JobID doesn't exist...

myEmps.AddLast(New clsEmployee("Lee", 45, 37000, 999))

myEmps.AddLast(New clsEmployee("Ann", 38, 41000, 305))

'All job assignments

myJobs.AddLast(New clsJobAssignment(104, "CEO", "5A"))

'Note that there are no CFO's in the employee work force

myJobs.AddLast(New clsJobAssignment(207, "CFO", "3A"))

myJobs.AddLast(New clsJobAssignment(200, "CIO", "4A"))

myJobs.AddLast(New clsJobAssignment(305, "Engineer", "3A"))

myJobs.AddLast(New clsJobAssignment(415, "Technician", "2A"))

'List people and the jobs they do...

LINQResults = From emps In myEmps

Join jobs In myJobs On emps.JobID Equals jobs.JobID

Select emps.Name, jobs.JobTitle

'Join is like an INNER JOIN in SQL

Debug.Write("List People and their jobs: ")

For Each emp In LINQResults

Debug.Write(emp.name & " - " & emp.jobtitle & ", ")

Next

Debug.WriteLine("")

'List jobs and the people that have them...

LINQResults = From jobs In myJobs

Join emps In myEmps On jobs.JobID Equals emps.JobID

Select jobs.JobTitle, emps.Name

'Join is like an INNER JOIN in SQL

Debug.Write("List Jobs and their people: ")

For Each job In LINQResults

Debug.Write(job.jobtitle & " - " & job.name & ", ")

Next

Debug.WriteLine("")

'Which jobs have no one assigned to them

LINQResults = From jobs In myJobs

Group Join emps In myEmps On jobs.JobID Equals emps.JobID

Into AllJobs = Group

Where AllJobs.Count = 0

Select jobs

'Group Join is like an LEFT OUTER JOIN in SQL

Debug.Write("List Jobs that haven't been assigned: ")

For Each job In LINQResults

Debug.Write(job.jobtitle & ", ")

Next

Debug.WriteLine("")

'Which people don't have a valid job record

LINQResults = From emps In myEmps

Group Join jobs In myJobs On emps.JobID Equals jobs.JobID

Into AllEmps = Group

Where AllEmps.Count = 0

Select emps

'Group Join is like an LEFT OUTER JOIN in SQL

Debug.Write("List Emps that haven't been assigned to a job: ")

For Each emp In LINQResults

Debug.Write(emp.name & ", ")

Next

Debug.WriteLine("")

End Sub

Here’s the output:

List People and their jobs: Bill - Technician, Sue - Engineer, Dave - CIO, Sara - CEO,

Ann - Engineer,

List Jobs and their people: CEO - Sara, CIO - Dave, Engineer - Sue, Engineer - Ann,

Technician - Bill,

List Jobs that haven't been assigned: CFO,

List Emps that haven't been assigned to a job: Lee,

I know that not everybody likes the Group Join logic, so we can attack the last two queries in a different manner: using the set operator Except:

'Chapter 9 - Program 11

Public Class clsEmployee

Public Name As String

Public Age As Integer

Public Salary As Single

Public JobID As Integer

Public Sub New(ByVal Name As String, ByVal Age As Integer,

ByVal Salary As Single, ByVal JobID As Integer)

Me.Name = Name

Me.Age = Age

Me.Salary = Salary

Me.JobID = JobID

End Sub

End Class

Public Class clsJobAssignment

Public JobID As Integer

Public JobTitle As String

Public PayGrade As String

Public Sub New(ByVal JobID As Integer, ByVal JobTitle As String,

ByVal PayGrade As String)

Me.JobID = JobID

Me.JobTitle = JobTitle

Me.PayGrade = PayGrade

End Sub

End Class

Sub Main()

'We're using a LinkedList now!

Dim myEmps As New LinkedList(Of clsEmployee)

Dim myJobs As New LinkedList(Of clsJobAssignment)

Dim LINQResults As Object

'All employees

myEmps.AddLast(New clsEmployee("Bill", 20, 45000, 415))

myEmps.AddLast(New clsEmployee("Sue", 35, 55000, 305))

myEmps.AddLast(New clsEmployee("Dave", 24, 37000, 200))

myEmps.AddLast(New clsEmployee("Sara", 22, 40000, 104))

'Note this person's JobID doesn't exist...

myEmps.AddLast(New clsEmployee("Lee", 45, 37000, 999))

myEmps.AddLast(New clsEmployee("Ann", 38, 41000, 305))

'All job assignments

myJobs.AddLast(New clsJobAssignment(104, "CEO", "5A"))

'Note that there are no CFO's in the employee work force

myJobs.AddLast(New clsJobAssignment(207, "CFO", "3A"))

myJobs.AddLast(New clsJobAssignment(200, "CIO", "4A"))

myJobs.AddLast(New clsJobAssignment(305, "Engineer", "3A"))

myJobs.AddLast(New clsJobAssignment(415, "Technician", "2A"))

'List jobs without people using Except

LINQResults = (From jobs In myJobs Select jobs.JobID).Except(From emps In myEmps

Select emps.JobID)

Debug.Write("List peopleless jobs: ")

For Each job In LINQResults

Debug.Write(job & " ")

Next

Debug.WriteLine("")

'List people without job titles

LINQResults = From emps2 In myEmps

Where ((From emps In myEmps Select emps.JobID).Except(From jobs In

myJobs Select jobs.JobID)).Contains(emps2.JobID)

Select emps2

Debug.Write("List titleless people: ")

For Each emp In LINQResults

Debug.Write(emp.name & " - " & emp.jobid)

Next

Debug.WriteLine("")

End Sub

Output:

List peopleless jobs: 207

List titleless people: Lee - 999

The first query brings back the jobs that are in the set of MyJobs.JobID but not in the set of myEmps.JobID. You may prefer this syntax over the GROUP JOIN logic. The second query is a little bit more complex. We are bringing back all employees that appear in the set of JobIDs that exist in the MyEmps set but not in the MyJobs set. In other words, find the employees who have jobs that don’t show up in the jobs list. Once you have them, get those JobID values. Now when you iterate across any employee who has a JobID that is in that inner list of missing JobID values, return that information. Yeah, you’ll probably have to think about that a little bit, but it’s not too bad. Bottom line – if you want to use LINQ, it’s like SQL and you may have to scratch your head a bit before you get things to work the way you want!

Now if the set logic approach is more appealing to you than the Group Join method, remember that you have the following operators to you:

|  |  |
| --- | --- |
| *Operator* | *Purpose* |
| Except | What’s in one set that’s not in the other |
| Intersect | Selects what’s in both sets |
| Union | Combines two sets together with no repeated values |
| Concat | Combines two sets together and will leave repeated values |

About the only other thing that I want you to know about LINQ is that it uses what is called Deferred Execution. Rather than running the query, getting the results and iterating over the results, LINQ does something different. Deferred Execution actual returns the results as the iteration is taking place, so rather than grabbing the result set at once and consuming a whole bunch of resources, LINQ brings values back over the iterator which makes debugging LINQ queries more difficult.