

# **.NET FUNDAMENTALS:** **GENERIC CLASSES AND** **METHODS**

Bill Sheldon

- Microsoft MVP - Visual Basic
- Senior Technical Analyst-Developer Rubio's Restaurants
- Author Office Business Application Development (WROX)
- Instructor UCSD Extension
- Author Professional Visual Basic (WROX)
- Author SQL Server Magazine

# Getting to Generics

- ▣ Like all languages the .NET languages came with the ability to save values in a variable...
  - ▣ *Let's take a walk from the creation of variables through generics*
  - ▣ *Not saying .Net introduced these or other structures – in fact I'm not even going to suggest that .NET introduced the underlying concepts behind generics...*
- ▣ Think back to the dawn of programming
  - This was a good starting point, I could save a value in something called 'x' and then reference it repeatedly.
  - The developers said “this is good but what if I want a set of related values?”

# Getting to Generics

- ▣ Languages next introduced the concept and implementation of arrays
  - Supported the creation of a set of related values
  - Had limitations – ex. Need to define size, no support for sparse data, allocated when defined etc.
  - The developers wanted more...
- ▣ Needed a way to dynamically add elements
- ▣ Had specialized concepts
  - Queue, Stack, List, Dictionary
  - Needed to constantly reinvent

# Getting to Generics

- ▣ Thus the concept and implementation of collections were included with .NET 1.0
  - Collections provided a way to dynamically add and remove elements from a set.
  - Customized for special needs like Queue, Stack and Dictionary, designed to accept any object
- ▣ It was good... but then someone realized there were typing issues:
  - Boxing (moving data in memory for no good reason)
  - Dogs and cats living together!

# Getting to Generics

- ▣ Needed a way to **specify** the type associated with a collection
- ▣ .NET 2.0 introduced Generics
- ▣ List<T> or List (Of T) tell the compiler that you are defining a collection which will only include objects of type T
- ▣ Eliminates Boxing
- ▣ Enforces Type Safety
- ▣ .NET ships with several classes that are implemented as generics

# Generics

- ▣ You can create custom Generics
  - Classes, Structures and Interfaces can all be defined to support Generics
  - Methods can also be defined to support Generics
- ▣ Generics essentially allow you to associate a template or blueprint with a generic definition
  - Don't think of List (Of T)/List<T> as a type
  - When you assign T you create a type
  - .NET can thus use all type based checking for your custom definition of List (Of T)
- ▣ The .NET framework exposes generics that you'll use 90+% of the time vs. rolling your own

# Generics in .NET

- ▣ .NET ships with several generics in the box
  - System.Collections.Generic is just one namespace
  - <http://msdn.microsoft.com/en-us/library/system.collections.generic.aspx>
  - List(T)
  - Dictionary(Tkey, Tvalue)
  - Stack(T) and Queue(T) (aka LIFO & FIFO)
  - Comparer(T)
- ▣ That same namespace also includes structures and interfaces

# Generics in .NET

Looking at using the Generic classes which ship with .NET

# Generics in .NET

- ▣ Best practice when working with Generic Collections – leverage the interfaces
  - Define your methods to accept an `ICollection<T>` instead of `List<T>` so that the actual implementation can vary. (or `IDictionary(Of TKey, TValue)`)
  - Go one better by using more specific interfaces like `IEnumerable(Of T)`, `ICollection(Of T)` and `IComparable(Of T)`
  - Will need to instantiate an `List(Of T)` or `Stack(Of T)` but only reference the underlying type when needed.

# Generics in .NET

- ▣ Collections aren't the only supplied generics
- ▣ System.Nullable
  - Used to create Integers and other value types which may be null
  - Good when working with a database
  - Nullable<T> or Nullable (Of T)
  - .HasValue and .Value are the public properties
  - .GetValueOrDefault(x) method
  - Both C# and VB support the ? Shortcut for defining a nullable value type
  - Dim x as Integer? Or Integer? y

# Generics in .NET

- ▣ LINQ also leverages Generics
- ▣ Table<T> or BindingList<T>
  - Don't know the specific type of 'table' that will be returned.
  - Led to the introduction of the 'var' type in C#
  - VB introduced Implicit type option – same thing var uses implicit typing
  - The results are retrieved until runtime but generics are early bound – var is still strongly typed.
- ▣ Seems like you are using a generic without ever explicitly declaring it

# Using Generics

- ▣ Reference the name of the generic and supply the necessary types

- ▣ C#:

```
List<int> intList = new List<int>();
```

- ▣ VB:

```
Dim intList As List(Of Integer) = _  
    New List(Of Integer)
```

- ▣ That pretty much covers it, you'll find you need to reference the full declaration for any parameter declarations (ex. ByVal x as List<T>)

# Using Generics

- ▣ Generics don't support polymorphism at the top level
  - I. E. I have a generic: List (Of JetFighters), I can't cast it to a generic: List (Of Planes)
  - Even though Planes is the parent class of JetFighters
- ▣ Generics do support polymorphism at the contents level
  - I have a generic list: List (Of Planes) which contains both JetFighters and AirLiners.

# Custom Classes

How to create your own custom Generic class implementations

# Creating your Own Generics

- ▣ Yes you can...

`Class Node (Of K, T)`

- ▣ That is the top level, but it doesn't provide much control if I need behavior from K or T
- ▣ Works fine for this class, but what of a container containing nodes
- ▣ Want to compare the different node to find one based on a `Get(K)` type request
  - Need a constraint to ensure K implements the `IComparable` interface

# Constraints

- ▣ Constraints allow you to ensure that a generic parameter meets certain minimum requirements
  - For example you could create a constraint for comparability (`IComparable`)
  - or for a class (`String`)
  - or even for something like a parameterless constructor (`New()`)
- ▣ Constraints should be used with care but allow you to have access to more than just base `Object` class methods in your implementation

# Constraints

- ▣ For C# constraints are defined with the where clause:

```
public class MyClass<T> where T:IComparable
```

- ▣ For VB constraints are defined with the As clause:

```
Public Class MyClass(Of T As MyBaseClass)
```

- ▣ Can have more than one parameter constrained

```
Public Class MyDictionary(Of K As  
IComparable, Of V As ICloneable)
```

# Creating a Custom Generic

- ▣ First let's declare a generic node that will be a simple key value pair class
  - Note there is already a key value pair generic available in the framework this is for illustration only.

```
Class Node (Of K, T)  
    Public Key As K  
    Public Item As T  
    Public NextNode As Node (Of K, T)  
End Class
```

# Creating a Custom Generic

- ▣ Define a constrained container

```
Public Class SinglyLinkedList(Of K
                                As IComparable(Of K), T)
    Dim m_Head As Node(Of K, T)
    ' Rest of the implementation
End Class
```

- ▣ Want to get cleaner?
  - declare the node class as a nested class of this generic class and encapsulate the implementation of your linked list privately – you'll need to modify Node's definition.

# Creating a Custom Generic

- ▣ A simple property definition to get an entry based on the key for this new class:

```
Public ReadOnly Property Item(ByVal key As K) As T
    Get
        Dim current As Node(Of K, T) = m_Head
        While current.NextNode IsNot Nothing
            If (current.Key.CompareTo(key) = 0) Then
                Exit While
            Else
                current = current.NextNode
            End If
        End While
        Return current.item
    End Get
End Property
```

# Generic Methods

Looking at using the Generic classes which ship with .NET

# Custom Generic Methods

- ▣ Allow you to specify the types associated with a method

- ▣ Consider

```
Public Boolean Compare (Object x, Object y)
```

- ▣ Is Y to be the same or a different type from X?
- ▣ Currently have to Check at run time... but I'd rather check at compile time
- ▣ Declare this as a generic and pass the type with the call

# Generic Methods

```
public Boolean Compare<P>(P x, P y)
{
    if (y.Equals(x))
        return true;
    return false;
}
```

- ▣ This method can then be called as:

```
Boolean res = Compare<int> (a, 2)
```

# Generic Methods

- ▣ Note that you are not required to define all parameters of type T
- ▣ Can also define method scoped variables of type T within your method
- ▣ By default the parameters are Object, unless you cast the parameters using the where clause.

`where T: IComparable`

- ▣ Need the As clause in VB  
(Of T As IComparable)

# Conclusion

- ▣ Now that we have generics its all better?
  - No we're developers – we want more
- ▣ If I have a set of common actions I want to do on different generic collections I don't have a way of just passing that collection
  - Yes even generics have limitations
- ▣ .NET 4.0 introduces Variance/Co-Variance
  - Allow you to create a method that accepts a List(Of T) or List<T> without defining T.
  - You can thus operate on different generics.

# Questions

- ▣ Additional resources
  - Generics FAQ: Fundamentals (on MSDN)
  - Generics FAQ: Tool Support (on MSDN)
  - Generics FAQ: Best Practices (On MSDN)
- ▣ My slides are posted on my blog:  
<http://www.nerdnotes.net/blog>