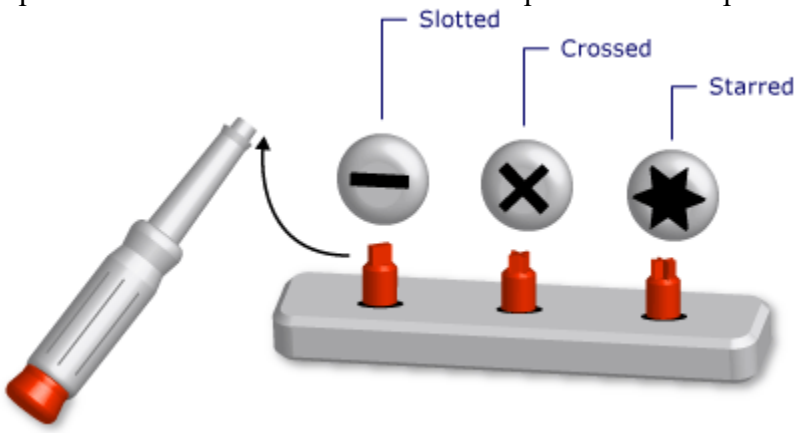
Generics in VB.net

A generic type is a single programming element that adapts to perform the same functionality for a variety of data types. When you define a generic class or procedure, you do not have to define a separate version for each data type for which you might want to perform that functionality. We can generic methods and generic classes in vb.net. Generics provide type **safety**, **faster execution** and **reuse of code** for those situations where our code needs to operate on different data. Generics are equivalent to templates in C++.



Generic Class:

The following example shows a skeleton definition of a generic class.

```
Public Class ClassName(Of T)
    Public Sub Do(a As T,b As T)
        Dim tempItem As T
    End Sub
End Class
```

In the preceding skeleton, `T` is a *type parameter*, that is, a placeholder for a data type that you supply when you declare the class. Elsewhere in your code, you can declare various versions of `ClassName` by supplying various data types for `T`. The following example shows two such declarations.
When we create new

Generic methods:

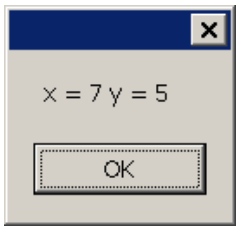
To develop the motivation for generics, create a new windows project called "Generics". Suppose we wanted to write a method to exchange two integers. For this purpose, we can add a class to the project called GenUtils with the following code in it.

```
Class GenUtil
    Public Shared Sub Exchange(ByRef a As Integer, ByRef b As Integer)
        Dim temp As Integer = a
        a = b
        b = temp
    End Sub
End Class
```

To test the exchange method, add a button to the form with a name of btnExchange with the following code in its handler.

```
Private Sub btnExchange_Click(ByVal sender As Object, ByVal e As EventArgs)
  Dim x As Integer = 5
  Dim y As Integer = 7
  GenUtil.Exchange(x, y)
  MessageBox.Show("x = " & x.ToString() & " y = " + y.ToString())
End Sub
```

If you build and run the program and click on the above button, you will see the following output.



Now what if we wanted to also be able to exchange two doubles, we can overload the exchange method in the GenUtil class by adding a method that takes two doubles and exchanges the values as shown below.

```
Class GenUtil
    Public Shared Sub Exchange(ByRef a As Integer, ByRef b As Integer)
        Dim temp As Integer = a
        a = b
        b = temp
    End Sub
    Public Shared Sub Exchange(ByRef a As Double, ByRef b As Double)
        Dim temp As Double = a
        a = b
        b = temp
    End Sub
End Class
```

Notice the difference in code in the above two functions is the first two lines in each function i.e.,

```
    Public Shared Sub Exchange(ByRef a As Integer, ByRef b As Integer)
           Dim temp As Integer = a
             ....

    Public Shared Sub Exchange(ByRef a As Double, ByRef b As Double)
           Dim temp As Double = a
             …
```

What if wanted a function also that can exchange two strings, and also that can exchange two student objects etc.. We can copy and paste the Exchange method and change the data type of the first two lines in the method. Even though this will work, it requires too many overloaded copies of the function in a class. Generics solve this problem by having

the developer declare only one method that can operate on any data type (i.e., generic data type), Modify the GenUtil class to include only one generic exchange method as:

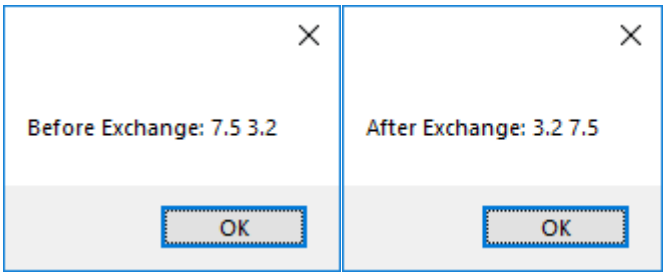
```
Class GenUtil
    Public Shared Sub Exchange(of T)(ByRef a As T, ByRef b As T)
        Dim temp As T = a
        a = b
        b = temp
    End Sub
End Class
```

The (of T) after the name of the function simply indicates a generic data type that will be decided by the compiler depending upon how the method will be invoked. For example, if some one calles the above method as:

```
    Private Sub btnExchange_Click(sender As Object, e As EventArgs) Handles
btnExchange.Click
        Dim x As Integer =10
        Dim y As Integer=20
        MessageBox.Show("Before Exchange: "+x.ToString()+" "+y.ToString())
        GenUtil.Exchange(x,y)
        MessageBox.Show("After Exchange: "+x.ToString()+" "+y.ToString())
    End Sub
```

T in the Exchange method in the GenUtil class will be treated as an int. On the other hand, if the Exchange method is invoked as:

```
        Dim x As Double =7.5
        Dim y As Double=3.2
        GenUtil.Exchange(x,y)
```



T in the Exchange method will become a double. The letter T itself is not a keyword. We can give the generic type any name we feel like. For example. We could have written the GenUtil class by replacing the T with MyType as:

```
Class GenUtil
    Public Shared Sub Exchange(of myType)(ByRef a As myType, ByRef b As myType)
        Dim temp As myType = a
        a = b
        b = temp
    End Sub
End Class
```

In the calling code, nothing special needs to done i.e., the call to a Generic method usually appears as if it was a normal method.

Generic Classes:

Add a class called MyGen to the project with the following code.

```
Class MyGen(Of T1, T2)
    Public Property A As T1
    Public Property B As T2
    Public Overrides Function ToString() As String
        Return A.ToString() & " : " & B.ToString()
    End Function
End Class
```

As you can see from the above code, it uses two generic types T1 and T2 that the class will use. The class then declares two data members of the types T1 and T2 respectively. To test creating objects of the above generic class, add a button called btnGenericClass to the form with the following test code in it.

```
    Private Sub btnGenericClass_Click(sender As Object, e As EventArgs) Handles
btnGenericClass.Click
        'Dim mg As MyGen(Of Integer, Single) = New MyGen(Of Integer, Single)()
        Dim mg = New MyGen(Of Integer, Single)()
        mg.A = 5
        mg.B = 3.75F
        MessageBox.Show(mg.ToString())
    End Sub
```

Notice how an object of a generic class is created i.e., within the (), the caller has to indicate the two data types the generic class expects to use.

```
Dim mg AS MyGen(Of Integer, Single) = new MyGen(Of Integer, Single)()
```

To demonstrate a more useful example of a generic class, we can create a class that can initialize an array of any reference type. If you recall, creating an array of a reference type is a two step process where in the first step, we create an array of references and in the second step, we create the objects in the array e.g., to create an array of Students, the code will look like:

```
Private Sub btnStudentArray_Click(sender As Object, e As EventArgs) Handles
btnStudentArray.Click
        Dim  stArr = New Student(5){}
        For i As Integer = 0 To stArr.Length
            stArr(i)=New Student()
        Next
    End Sub
```

If we were to create an array of Employees, the code will look as:

```
        Dim  stArr = New Employee(5){}
        For i As Integer = 0 To stArr.Length
            stArr(i)=New Employee()
        Next
```

Since creating an array of reference types is needed quite often, we could create a generic class with a method in it that allows proper creating of an array of any reference type. To see this, add a class to the project called GenArr with the following code in it.

```
Class GenArr
    Public Shared Function InitArray(Of T As New)(ByVal size As Integer) As T()
        Dim arr As T() = New T(size - 1) {}
        For i As Integer = 0 To size - 1
            arr(i) = New T()
        Next
```

```
        Return arr
    End Function
End Class
```

The statement `Of T As New` in the above function indicates that the T has to be creatable with a new keyword (as in creating an object of a class using the new keyword).

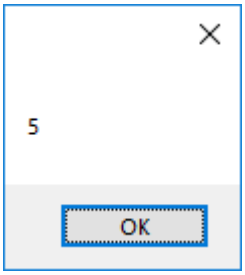
Add a class called Student to the project with the following code in it.

```
Public Class Student
    Public Property FirstName As String
    Public Property LastName As String
    Public Property Id As Integer
    Public Property Test1Score As Integer
    Public Property Test2Score As Integer
    Public Overrides Function ToString() As String
        Return FirstName & " " & LastName & " " + Id.ToString() & " " +
Test1Score.ToString() & " " + Test2Score.ToString()
    End Function
End Class
```

To the test the above generic method, add a button called btnInitArray with the following code in its handler.

```
private void btnInitArray_Click(object sender, EventArgs e)
{
    Dim  stArr =GenArr.InitArray(Of Student)(5)' New Student(5){}
    MessageBox.Show(STArr.Length.ToString())
}
```

Running he program and clicking on the above button produces the following output.



What is we wanted to be able to find the maximum test score 1 from an array of Students, or finding an Emplyee with the highest salaray from an array of employees. To achieve this, we can develop a generic method that can determine the maximum of any generic array of objects. However, for this method to be able the determine the maxmum, the class whose array is being created must provide an implementation of the IComparable interface. Modify the GenArr class to include a generic FindMax method as shown below.

```
Public Class GenArr
    Public Shared Function InitArray(Of T As New)(ByVal size As Integer) As T()
        Dim arr As T() = New T(size - 1) {}
        For i As Integer = 0 To size - 1
            arr(i) = New T()
```

```
        Next
        Return arr
    End Function
    Public Shared Function FindMax(Of T As IComparable)(ByVal arr As T()) As T
        Dim max As T = arr(0)
        For i As Integer = 1 To arr.Length-1
            If arr(i).CompareTo(max) = 1 Then max = arr(i)
        Next
        Return max
    End Function
End Class
```

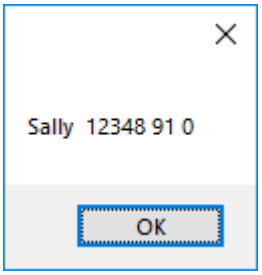
If we wanted to be able to determine the Student with the maximum TestScore1 from an array of Students, the Student class must implement the **IComparable interface** as shown below.

```
Public Class Student
    Implements IComparable
    Public Property FirstName As String
    Public Property LastName As String
    Public Property Id As Integer
    Public Property Test1Score As Integer
    Public Property Test2Score As Integer
 Public Overrides Function ToString() As String
        Return FirstName & " " & LastName & " " + Id.ToString() & " " +
Test1Score.ToString() & " " + Test2Score.ToString()
   End Function

  Public Function CompareTo(obj As Object) As Integer Implements
IComparable.CompareTo
        Dim res As Integer = 0
        Dim st As Student = Nothing
        If TypeOf obj Is Student Then
            st = CType(obj, Student)
            res = Me.Test1Score.CompareTo(st.Test1Score)
        End If
        Return res
  End Function
End Class
```

To test the FindMax generic method, add a button to the form called btnFindMaxScoreStudent with the following code in it.

```
  Private Sub btnStudentArray_Click(sender As Object, e As EventArgs) Handles
btnStudentArray.Click
        Dim stArr = GenArr.InitArray(of Student)(3)
        stArr(0).Id = 12345
        stArr(0).FirstName = "Bill"
        stArr(0).Test1Score = 83
        stArr(1).Id = 12348
        stArr(1).FirstName = "Sally"
        stArr(1).Test1Score = 91
        stArr(2).Id = 12346
        stArr(2).FirstName = "Mark"
        stArr(2).Test1Score = 85
        Dim maxScoreStudent = GenArr.FindMax(of Student)(stArr)
        MessageBox.Show(maxScoreStudent.ToString())
```

```
    End Sub
```



Generic Classes and Interfaces in .Net Library:

There are many generic classes and interfaces in the library especially for those cases where the code may need to operate on different data. For example, for sorting purposes, there are generic equivalents of the **IComparable** and **IComparer** interfaces.

Exercise: Convert the Icomparable and Icomparer implementations of the Student class to generic versions of these interfaces.
Solution:
The modified Student class that implements IComparable(of T)appears as:

```
 Public Class Student
    Implements IComparable(Of Student)
    Public Property FirstName As String
    Public Property LastName As String
    Public Property Id As Integer
    Public Property Test1Score As Integer
    Public Property Test2Score As Integer
    Public Overrides Function ToString() As String
        Return FirstName & " " & LastName & " " + Id.ToString() & " " +
Test1Score.ToString() & " " + Test2Score.ToString()
    End Function
    Public Function CompareTo(ByVal other As Student) As Integer Implements
IComparable(Of Student).CompareTo
        Return Me.Test1Score.CompareTo(other.Test1Score)
    End Function
End Class
```

Notice how efficient the CompareTo function in the above implementation of IComparable(of Student) is as compared the previous non generic IComparable that required type checking and conversion at run time.

For the FindMax generic function to work with the new Student class IComparable<Student< implementation, we will need to modify the GenArr class as (modification needed is shown in bold):

```
class GenArr
{
 Public Shared Function FindMax(Of T As IComparable(Of T))(ByVal arr As T()) As T
        Dim max As T = arr(0)
        For i As Integer = 1 To arr.Length
            If arr(i).CompareTo(max) = 1 Then max = arr(i)
          Next
        Return max
```

```
    End Function
}
```

Implementation of Icomparer(of T) for Student class:

Add a class called MyEnums with the following code in it.

```
       Public Enum SORTFIELD As Integer
        FIRSTNAME
        LASTNAME
        ID
        TEST1SCORE
        TEST2SCORE
    End Enum

    Public  Enum SORTDIR As Integer
        ASC
        DESC
    End Enum
```

Add a class called StudentComparer with the following code in it. This class implements Icomparer<Student> interface. Notice the Compre method in the interface does not require any type checking and conversion of the two parameters as was needed in the non generic IComparer interface implementation.

```
Class StudentComparer
    Implements IComparer(Of Student)
    Public Property SortFields As SORTFIELD = SORTFIELD.ID
    Public Property SortDirs As SORTDIR = SORTDIR.ASC
    Public Function Compare(ByVal x As Student, ByVal y As Student) As Integer
         Implements IComparer(Of Student).Compare
        Dim res As Integer = x.CompareTo(y, SortFields)
        If SortDirs = SORTDIR.DESC Then res = -1 * res
        Return res
    End Function
End Class
```

Modify the Student class to provide the CompareTo(Student st, SORTFIELD sField) method as:

```
Public Class Student
    Implements IComparable(of Student)
    Public Property FirstName As String
    Public Property LastName As String
    Public Property Id As Integer
    Public Property Test1Score As Integer
    Public Property Test2Score As Integer

    Public Overrides Function ToString() As String
        Return FirstName & " " & LastName & " " + Id.ToString() & " " +
Test1Score.ToString() & " " + Test2Score.ToString()
    End Function
    Public Function CompareTo(ByVal st As Student, ByVal sField As SORTFIELD) As
Integer
        Dim res As Integer = 0
        Select Case sField
            Case SORTFIELD.FIRSTNAME
                res = FirstName.CompareTo(st.FirstName)
            Case SORTFIELD.LASTNAME
```

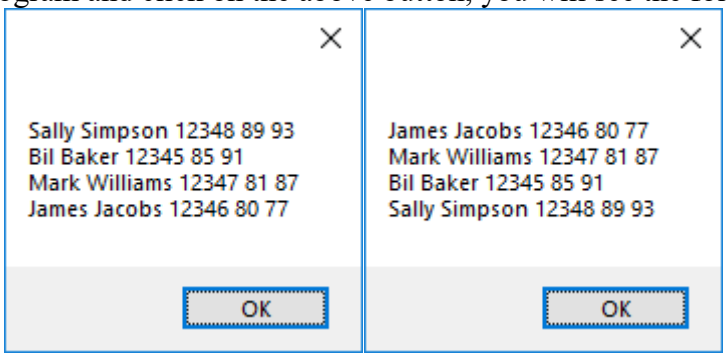
```
                res = LastName.CompareTo(st.LastName)
            Case SORTFIELD.ID
                res = Me.Id.CompareTo(st.Id)
            Case SORTFIELD.TEST1SCORE
                res = Me.Test1Score.CompareTo(st.Test1Score)
            Case SORTFIELD.TEST2SCORE
                res = Me.Test2Score.CompareTo(st.Test2Score)
        End Select

        Return res
    End Function
    Public Function CompareTo(ByVal other As Student) As Integer Implements
IComparable(Of Student).CompareTo
        Return Me.Test1Score.CompareTo(other.Test1Score)
    End Function
End Class
```

To test the IComparer (of T) implementation, add a button to the form with a name of btnComparerGeneric with the following code in the button handler.

```
Private Sub btnIComparerGeneric_Click(sender As Object, e As EventArgs) Handles
btnIComparerGeneric.Click
        Dim STList As List(Of Student) = New List(Of Student)
        Dim s1 As Student = New Student With {
            .FirstName = "Bil",
            .LastName = "Baker",
            .Test1Score = 85,
            .Test2Score = 91,
            .Id = 12345
        }
        STList.Add(s1)
        Dim s2 As Student = New Student() With {
            .FirstName = "Sally",
            .LastName = "Simpson",
            .Test1Score = 89,
            .Test2Score = 93,
            .Id = 12348
        }
        STList.Add(s2)
        Dim s3 As Student = New Student() With {
            .FirstName = "Mark",
            .LastName = "Williams",
            .Test1Score = 81,
            .Test2Score = 87,
            .Id = 12347
        }
        STList.Add(s3)
        Dim s4 As Student = New Student() With {
            .FirstName = "James",
            .LastName = "Jacobs",
            .Test1Score = 80,
            .Test2Score = 77,
            .Id = 12346
        }
        STList.Add(s4)
        Dim sc As StudentComparer2 = New StudentComparer2
        sc.SortFields = SORTFIELD.TEST2SCORE
        sc.SortDirs = SORTDIR.DESC
```

```
        STList.Sort(sc)
        Dim out1 As String = ""
        For Each st As Student In STList
            out1 = (out1 + (st.ToString + "" & vbLf))
        Next
        MessageBox.Show(out1)
        sc.SortFields = SORTFIELD.TEST2SCORE
        sc.SortDirs = SORTDIR.ASC
        STList.Sort(sc)
        out1 = ""
        For Each st As Student In STList
            out1 = (out1 + (st.ToString + "" & vbLf))
        Next
        MessageBox.Show(out1)
    End Sub
```

If you run the program and click on the above button, you will see the following output.



The above test code uses List which is generic equivalent of the ArrayList class. List is preferred over ArrayList as it gives us type safety at compile time, and thus results in faster execution as well. For example, if you declare a List of Students as:

```
Dim STList = new List(of Student)();
```

If you try to add a type other than Student to the STList (that is not derived from Student), you will get a compile time error.

Just like the List is the generic equivalent of the ArrayList class, similarly Dictionary is the generic equivalent of the Hashtable class in .Net library.
To demonstrate the use of Dictionary, add a button to the form called btnDictionary with the following code in its handler.

```
            Private Sub btnDictionary_Click(sender As Object, e As EventArgs)
Handles btnDictionary.Click
        Dim DTable As Dictionary(Of Integer, Student2) = New Dictionary(Of
Integer, Student2)

        Dim s1 As Student2 = New Student2 With {
                .FirstName = "Bil",
                .LastName = "Baker",
                .Test1Score = 85,
                .Test2Score = 91,
                .Id = 12345
                }
```

```
    DTable.Add(s1.Id, s1)
    Dim s2 As Student2 = New Student2() With {
            .FirstName = "Sally",
            .LastName = "Simpson",
            .Test1Score = 89,
            .Test2Score = 93,
            .Id = 12365
            }
    DTable.Add(s2.Id, s2)
    Dim id As Integer = 12365
    Try
        Dim st As Student2 = DTable(id)
        MessageBox.Show(st.ToString)
    Catch ex As KeyNotFoundException
        MessageBox.Show("Student does not exist")
    End Try
End Sub
```