CPSC 501 – Assignment 4
Generics, Extension Methods, Exceptions in C#

We can generic methods and generic classes in C#. 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 methods:

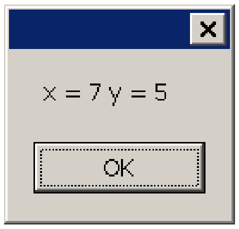
To develop the motivation for generics, create a new windows project called "Generics". Suppose we wanted to write a method to exchange two integers. For this purpose, we can add a class to the project called GenUtils with the following code in it.

```
    class GenUtil
    {
        public static void Exchange<T>(ref int a, ref int b)
        {
            int temp = a;
            a = b;
            b = temp;

        }
    }
```

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

```
        private void btnExchange_Click(object sender, EventArgs e)
        {
            int x = 5;
            int y = 7;
            GenUtil.Exchange(ref x, ref y);
            MessageBox.Show("x = " + x.ToString() + " y = " +
y.ToString());
        }
```

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


x = 7 y = 5

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 static void Exchange(ref int a, ref int b)
        {
            int temp = a;
            a = b;
            b = temp;
        }

        public static void Exchange(ref double a, ref double b)
        {
            double temp = a;
            a = b;
            b = temp;
        }
    }
```

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

```
        public static void Exchange(ref int a, ref int b)
        {
            int temp = a;
            ....

        public static void Exchange(ref double a, ref double b)
        {
            double temp = 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 static void Exchange<T>(ref T x, ref T y)
        {
            T temp = x;
            x = y;
            y = temp;
        }
    }
}
```

The <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:

```
            int x = 5;
            int y = 7;
            GenUtil.Exchange(ref x, ref y);
```

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:

```
        double p = 5.8;
        int q = 7.3;
        GenUtil.Exchange(ref p, ref q);
```

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 static void Exchange<MyType>(ref MyType x, ref MyType y)
        {
            MyType temp = x;
            x = y;
            y = temp;
        }
    }
}
```

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<T1,T2>
    {
        T1 a;
        public T1 A
        {
            get { return a; }
            set { a = value; }
        }

        T2 b;
        public T2 B
        {
            get { return b; }
            set { b = value; }
        }

        public override string ToString()
        {
            return a.ToString() + " : " + b.ToString();
        }
    }
```

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 obects of the above generic class, add a button called btnGenericClass to the form with the following test code in it.

```
private void btnGenericClass_Click(object sender, EventArgs e)
{
    MyGen<int, float> mg = new MyGen<int, float>();
    mg.A = 5;
    mg.B = 3.75f;
    MessageBox.Show(mg.ToString());
}
```

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.

```
MyGen<int, float> mg = new MyGen<int, float>();
```

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 void btnStudentArray_Click(object sender, EventArgs e)
 {
    Student[] STArr = new Student[5];
    for (int i = 0; i < STArr.Length; i++)
        STArr[i] = new Student();
 }
```

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

```
Employee[] EArr = new Employee[5];
for (int i = 0; i < EArr.Length; i++)
    EArr[i] = new Employee();
```

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<T>
{
    public static T[] InitArray<T>(int size)
        where T : new()
    {
        T[] Arr = new T[size];
        for (int i = 0; i < size; i++)
            Arr[i] = new T();
        return Arr;
    }
}
```

The statement `where T : 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).

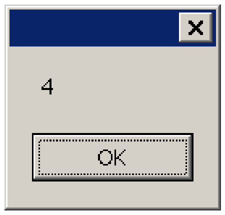
Add a class called Student to the project with the following code in it.

```
    public class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Id { get; set; }
        public int Test1Score { get; set; }
        public int Test2Score { get; set; }

        public override string ToString()
        {
             return FirstName + " " +
                 LastName + " " + Id.ToString() +
                 " " + Test1Score.ToString() +
                 " " + Test2Score.ToString();
        }
    }
```

To 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)
        {
            Student[] STArr = GenArr.InitArray<Student>(4);
            MessageBox.Show(STArr.Length.ToString());
        }
```

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


4

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 maximum, 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.

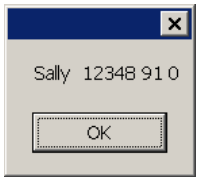
```
    class GenArr
    {
        public static T FindMax<T>(T[] Arr)
            where T : IComparable
        {
            T max = Arr[0];
            if (Arr[1].CompareTo(max) == 1)
                max = Arr[1];
            return max;
        }

        public static T[] InitArray<T>(int size)
            where T : new()
        {
            T[] Arr = new T[size];
            for (int i = 0; i < size; i++)
                Arr[i] = new T();
            return Arr;
        }
    }
```

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 : IComparable
    {
            ….

        #region IComparable Members

        public int CompareTo(object obj)
        {
            int res = 0;
            Student st = null;
            if (obj is Student)
            {
                st = (Student)obj;
                res = this.Test1Score.CompareTo(st.Test1Score);
            }
            return res;
        }

        #endregion
    }
```

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

```
  private void btnFindMaxScoreStudent_Click(object sender, EventArgs e)
        {
            Student [] STArr = GenArr.InitArray<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;

            Student maxScoreStudent = GenArr.FindMax<Student>(STArr);
            MessageBox.Show(maxScoreStudent.ToString());
        }
```



```
Sally  12348 91 0
```

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<T> appears as:

```
public class Student : IComparable<Student>
 {
     public string FirstName { get; set; }
     public string LastName { get; set; }
     public int Id { get; set; }
     public int Test1Score { get; set; }
     public int Test2Score { get; set; }

     public override string ToString()
     {
          return FirstName + " " +
             LastName + " " + Id.ToString() +
             " " + Test1Score.ToString() +
             " " + Test2Score.ToString();
     }


     #region IComparable<Student> Members

     public int CompareTo(Student other)
     {
         return this.Test1Score.CompareTo(other.Test1Score);
     }

     #endregion
 }
```

Notice how efficient the CompareTo function in the above implementation of IComparable<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 static T FindMax<T>(T[] Arr)
         where T : IComparable<T>
     {
         T max = Arr[0];
         if (Arr[1].CompareTo(max) == 1)
             max = Arr[1];
         return max;
     }
```

Implementation of IComparer<T> for Student class:

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

```
    public enum SORTFIELD : int
    {
        FIRSTNAME,
        LASTNAME,
        ID,
        TEST1SCORE,
        TEST2SCORE
    }

    public enum SORTDIR : int
    {
        ASC,
        DESC
    }
```

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 : IComparer<Student>
    {
        SORTFIELD sortField = SORTFIELD.ID;
        public SORTFIELD SortField
        {
            get { return sortField; }
            set { sortField = value; }
        }

        SORTDIR sortDir = SORTDIR.ASC;
        public SORTDIR SortDir
        {
            get { return sortDir; }
            set { sortDir = value; }
        }
        #region IComparer<Student> Members

        public int Compare(Student x, Student y)
        {
            int res = x.CompareTo(y,sortField);
            if (sortDir == SORTDIR.DESC)
                res = -1 * res;
            return res;
        }

        #endregion
    }
```

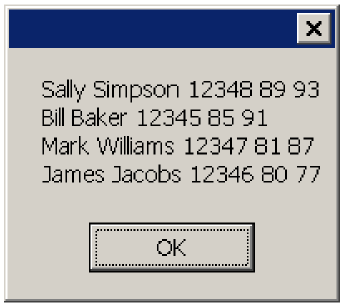
Modify the Student class to provide the CompareTo(Student st, SORTFIELD sField) method as:

```
    public class Student : IComparable<Student>
    {
        …
        public int CompareTo(Student st, SORTFIELD sField)
        {
            int res = 0;
            switch (sField)
            {
                case SORTFIELD.FIRSTNAME:
                    res = this.FirstName.CompareTo(st.FirstName);
                    break;
                case SORTFIELD.LASTNAME:
                    res = this.LastName.CompareTo(st.LastName);
                    break;
                case SORTFIELD.ID:
                    res = this.Id.CompareTo(st.Id);
                    break;
                case SORTFIELD.TEST1SCORE:
                    res = this.Test1Score.CompareTo(st.Test1Score);
                    break;
                case SORTFIELD.TEST2SCORE:
                    res = this.Test2Score.CompareTo(st.Test2Score);
                    break;
            }
            return res;
        }
    }
```

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

```
private void btnIComparerGeneric_Click(object sender, EventArgs e)
        {
            List<Student> STList = new List<Student>();
            // List is the generic equivalent of ArrayList class.

            Student s1 = new Student
            {
                FirstName = "Bill",
                LastName = "Baker",
                Test1Score = 85,
                Test2Score = 91,
                Id = 12345
            };
            STList.Add(s1);

            Student s2 = new Student
            {
                FirstName = "Sally",
                LastName = "Simpson",
                Test1Score = 89,
                Test2Score = 93,
                Id = 12348
            };
            STList.Add(s2);

            Student s3 = new Student
            {
                FirstName = "Mark",
                LastName = "Williams",
                Test1Score = 81,
                Test2Score = 87,
                Id = 12347
            };
            STList.Add(s3);

            Student s4 = new Student
            {
                FirstName = "James",
                LastName = "Jacobs",
                Test1Score = 80,
                Test2Score = 77,
                Id = 12346
            };
            STList.Add(s4);

            StudentComparer sc = new StudentComparer();
            sc.SortField = SORTFIELD.TEST2SCORE;
            sc.SortDir = SORTDIR.DESC;
            STList.Sort(sc); // will use IComparer to sort

            string out1 = "";
            foreach (Student st in STList)
                out1 += st.ToString() + "\n";

            MessageBox.Show(out1);
        }
```

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



```
Sally Simpson 12348 89 93
Bill Baker 12345 85 91
Mark Williams 12347 81 87
James Jacobs 12346 80 77
```

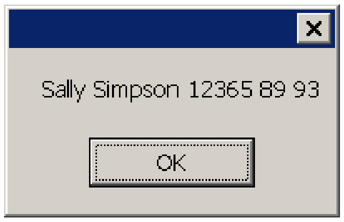
The above test code uses List which is generic equivalent of the ArrayList class. List is preferred over ArrayList as it gives us type safety at compile time, and thus results in faster execution as well. For example if you declare a List of Students as:

```
        List<Student> STList = new List<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 void btnDictionary_Click(object sender, EventArgs e)
    {
        // Dictionary is the generic equivalent of Hashtable
        Dictionary<int, Student> DTable = new Dictionary<int, Student>();
        Student s1 = new Student
        {
            FirstName = "Bill",
            LastName = "Baker",
            Id = 12337,
            Test1Score = 87,
            Test2Score = 91
        };
        DTable.Add(s1.Id, s1);
        Student s2 = new Student
        {
            FirstName = "Sally",
            LastName = "Simpson",
            Id = 12365,
            Test1Score = 89,
            Test2Score = 93
        };
        DTable.Add(s2.Id, s2);

        // lookup a student
        int id = 12365;
        try
        {
            Student st = DTable[id];
            MessageBox.Show(st.ToString());
        }
        catch (KeyNotFoundException)
        {
            MessageBox.Show("Student does not exist");
        }
    }
```



```
Sally Simpson 12365 89 93
```