

Objectives:

- Generics
- Anonymous methods
- Partial classes
- Nullable type

1. Use generic list to store your own class

```
using System.Collections.Generic;
using System;

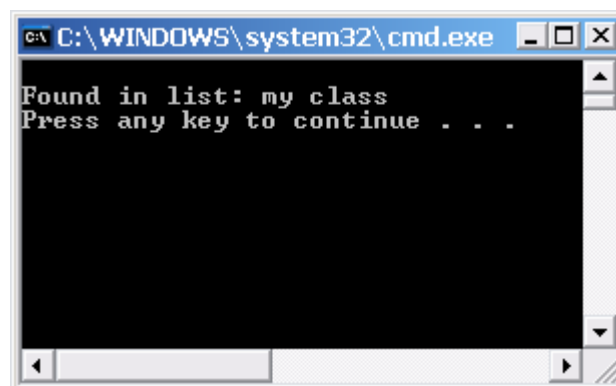
class MainClass
{
    public static void Main(string[] args)
    {
        List<MyClass> list = new List<MyClass>();

        list.Add(new MyClass());

        MyClass ass2 = list[0];

        Console.WriteLine("\nFound in list: {0}", ass2);
    }
}

class MyClass
{
    public override string ToString()
    {
        return "my class";
    }
}
```



2. Store user-defined Objects in a List collection

```
using System;
using System.Collections.Generic;

class Product
{
    string name;
    double cost;
    int onhand;

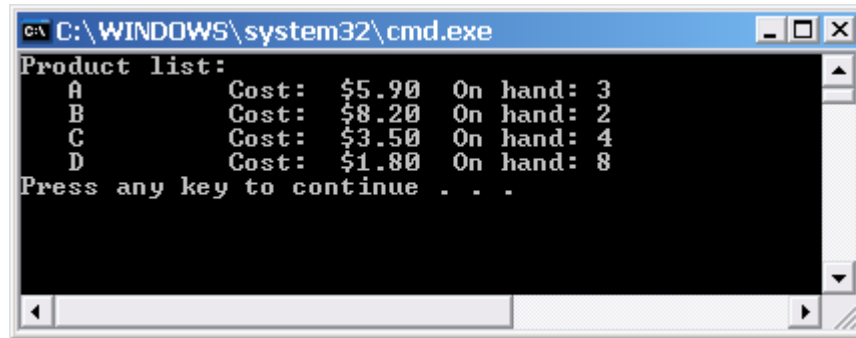
    public Product(string n, double c, int h)
    {
        name = n;
        cost = c;
        onhand = h;
    }

    public override string ToString()
    {
        return
            String.Format("{0,-10}Cost: {1,6:C}  On hand: {2}",
                           name, cost, onhand);
    }
}

class MainClass
{
    public static void Main()
    {
        List<Product> inv = new List<Product>();

        // Add elements to the list
        inv.Add(new Product("A", 5.9, 3));
        inv.Add(new Product("B", 8.2, 2));
        inv.Add(new Product("C", 3.5, 4));
        inv.Add(new Product("D", 1.8, 8));

        Console.WriteLine("Product list:");
        foreach (Product i in inv)
        {
            Console.WriteLine("    " + i);
        }
    }
}
```



C:\WINDOWS\system32\cmd.exe

```
Product list:
A      Cost: $5.90  On hand: 3
B      Cost: $8.20  On hand: 2
C      Cost: $3.50  On hand: 4
D      Cost: $1.80  On hand: 8
Press any key to continue . . .
```

3. Generic Interface

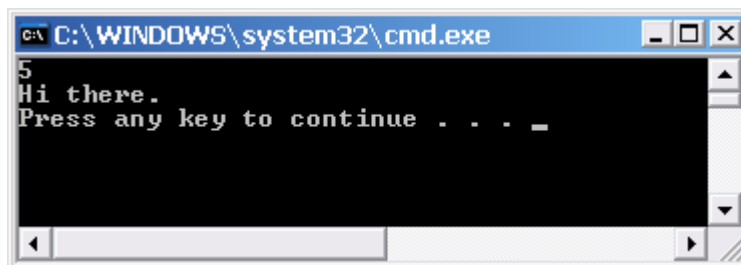
```
using System;
using System.Collections.Generic;

interface GenericInterface<T>
{
    T getValue(T tValue);
}

class MyClass<T> : GenericInterface<T>
{
    public T getValue(T tValue)
    {
        return tValue;
    }
}

class MainClass
{
    static void Main()
    {
        MyClass<int> intObject = new MyClass<int>();
        MyClass<string> stringObject = new MyClass<string>();

        Console.WriteLine("{0}", intObject.getValue(5));
        Console.WriteLine("{0}", stringObject.getValue("Hi
there."));
    }
}
```



C:\WINDOWS\system32\cmd.exe

```
5
Hi there.
Press any key to continue . . .
```

4. Generic class

```
using System;

class TwoGen<T, V>
{
    T ob1;
    V ob2;

    public TwoGen(T o1, V o2)
    {
        ob1 = o1;
        ob2 = o2;
    }

    public void showTypes()
    {
        Console.WriteLine("Type of T is " + typeof(T));
        Console.WriteLine("Type of V is " + typeof(V));
    }

    public T getT()
    {
        return ob1;
    }

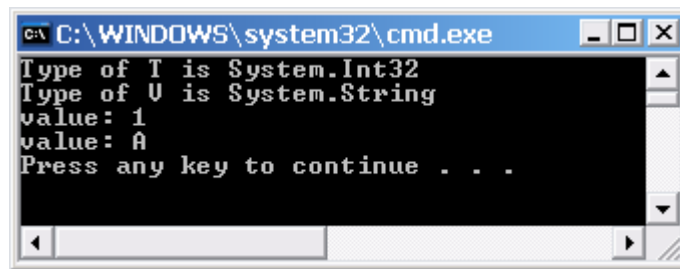
    public V getV()
    {
        return ob2;
    }
}

class MainClass
{
    public static void Main()
    {
        TwoGen<int, string> tgObj = new TwoGen<int, string>(1,
        "A");

        tgObj.showTypes();

        int v = tgObj.getT();
        Console.WriteLine("value: " + v);

        string str = tgObj.getV();
        Console.WriteLine("value: " + str);
    }
}
```



5. Partial Type

```
public partial class CoOrds
{
    private int x;
    private int y;

    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class CoOrds
{
    public void PrintCoOrds()
    {
        System.Console.WriteLine("CoOrds: {0},{1}", x, y);
    }
}

class TestCoOrds
{
    static void Main()
    {
        CoOrds myCoOrds = new CoOrds(10, 15);
        myCoOrds.PrintCoOrds();
    }
}
```

6. Partial Type

Step 1: Create new project name PartialTypes

Step 2: Create file name CharTypesPrivate.cs vaf type below code

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace PartialClassesExample
{
    // The partial keyword allows additional methods, fields, and
    // properties of this class to be defined in other .cs files.
    // This file contains private methods defined by CharValues.
    partial class CharValues
    {
        private static bool IsAlphabetic(char ch)
        {
            if (ch >= 'a' && ch <= 'z')
                return true;
            if (ch >= 'A' && ch <= 'Z')
                return true;
            return false;
        }

        private static bool IsNumeric(char ch)
        {
            if (ch >= '0' && ch <= '9')
                return true;
            return false;
        }
    }
}
```

Step 3: Create new file named CharTypesPublic.cs and type below code

```
using System;
using System.Collections.Generic;
using System.Text;

namespace PartialClassesExample
{
    // The partial keyword allows additional methods, fields, and
    // properties of this class to be defined in other .cs files.
    // This file contains the public methods defined by
    CharValues.
    partial class CharValues
    {
        public static int CountAlphabeticChars(string str)
        {
            int count = 0;
            foreach (char ch in str)
            {
                // IsAlphabetic is defined in CharTypesPrivate.cs
                if (IsAlphabetic(ch))
                    count++;
            }
        }
    }
}
```

```
    }  
    return count;  
}  
public static int CountNumericChars(string str)  
{  
    int count = 0;  
    foreach (char ch in str)  
    {  
        // IsNumeric is defined in CharTypesPrivate.cs  
        if (IsNumeric(ch))  
            count++;  
    }  
    return count;  
}  
}  
}
```

Step 4: Create file named: PartialTypes.cs, type below code

```
using System;  
using System.Collections.Generic;  
using System.Text;  
  
namespace PartialClassesExample  
{  
    class PartialClassesMain  
    {  
        static void Main(string[] args)  
        {  
            if (args.Length != 1)  
            {  
                Console.WriteLine("One argument required.");  
                return;  
            }  
  
            // CharValues is a partial class -- two of its  
            // methods  
            // are defined in CharTypesPublic.cs and two are  
            // defined  
            // in CharTypesPrivate.cs.  
            int aCount =  
CharValues.CountAlphabeticChars(args[0]);  
            int nCount = CharValues.CountNumericChars(args[0]);  
  
            Console.WriteLine("The input argument contains {0}  
alphabetic and {1} numeric characters", aCount, nCount);  
        }  
    }  
}
```

```
}
```

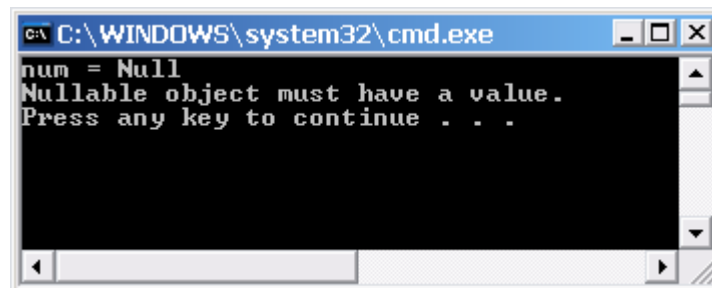
Step 5: In command line prompt, use cd command to change to PartialTypes directory

Step 6: Type the following:

```
csc PartialTypes.cs  
PartialTypes
```

7. Nullalbe Type

```
class NullableExample  
{  
    static void Main()  
    {  
        int? num = null;  
        if (num.HasValue == true)  
        {  
            System.Console.WriteLine("num = " + num.Value);  
        }  
        else  
        {  
            System.Console.WriteLine("num = Null");  
        }  
  
        //y is set to zero  
        int y = num.GetValueOrDefault();  
  
        // num.Value throws an InvalidOperationException if  
        num.HasValue is false  
        try  
        {  
            y = num.Value;  
        }  
        catch (System.InvalidOperationException e)  
        {  
            System.Console.WriteLine(e.Message);  
        }  
    }  
}
```



8. HasValue property

```
// This code example demonstrates the Nullable<T>.HasValue
// and Value properties.

using System;

class Sample
{
    public static void Main()
    {
        DateTime? myNow;

        // Assign the current date and time to myNow then
        display its value.
        myNow = DateTime.Now;
        Display(myNow, "1) ");

        // Assign null (Nothing in Visual Basic) to myNow then
        display its value.
        myNow = null;
        Display(myNow, "2) ");
    }

    // Display the date and time.
    public static void Display(DateTime? displayDateTime, string
title)
    {
        // If a value is defined for the displayDatetime
        argument, display its value; otherwise,
        // display that no value is defined.
        Console.Write(title);
        if (displayDateTime.HasValue == true)
            Console.WriteLine("The date and time is {0:F}.",
displayDateTime.Value);
        else
            Console.WriteLine("The date and time is not
defined.");
    }
}
```

Do It Yourself

8.1. Do the workshop 14, 15 in the CD

8.2. Do ACTCSharp_Module14_Assignment.pdf in CD

8.3. Do ACTCSharp_Module15_Assignment.pdf in CD

8.4. Do the exercise 7.3 again in generics.

`static int GreaterCount(List<double> list, double min)`

8.5. Create a new source file GenericDelegate.cs and declare a generic delegate type `Action<T>` that has return type `void` and takes as argument a `T` value. This is a generalization of yesterday's delegatetype `IntAction`.

Declare a class that has a method

`static void Perform<T>(Action<T> act, params T[] arr) { ... }`

This method should apply the delegate `act` to every element of the array `arr`. Use the `foreach` statement when implementing method `Perform<T>`.

8.6. The purpose of this exercise is to illustrate computations with nullable types over simple types such as `double`.

To do this, implement methods that work like SQL's aggregate functions. We don't have a database query at hand, so instead let each method take as argument an `IEnumerable<double?>`, that is, as sequence of nullable doubles:

- `Count` should return an `int` which is the number of non-null values in the enumerable.
- `Min` should return a `double?` which is the minimum of the non-null values, and which is null if there are no non-null values in the enumerable.
- `Max` is similar to `Min` and there is no point in implementing it.
- `Avg` should return a `double?` which is the average of the non-null values, and which is null if there are no non-null values in the enumerable.
- `Sum` should return a `double?` which is the sum of the non-null values, and which is null if there are no non-null values. (Actually, this is weird: Mathematically the sum of no elements is 0.0, but the SQL designers decided otherwise. This design mistake will also make your implementation of `Sum` twice as complicated as necessary: 8 lines instead of 4). When/if you test your method definitions, note that null values of any type are converted to the empty string when using `String.Format` or `Console.WriteLine`.

8.7. Get the file `IntList.cs` below. The file declares some delegate types:

`public delegate bool IntPredicate(int x);`

`public delegate void IntAction(int x);`

The file further declares a class `IntList` that is a subclass of .Net's `List<int>` class. Class `IntList` uses the delegate types in two methods that take a delegate as argument:

- `list.Act(f)` applies delegate `f` to all elements of `list`.
- `list.Filter(p)` creates a new `IntList` containing those elements `x` from `list` for which `p(x)` is true.

Add code to the file's `Main` method that creates an `IntList` and calls the `Act` and `Filter` methods on that list and various anonymous delegate expressions. For instance, if `xs` is an `IntList`, you can print all its elements like this:

`xs.Act(Console.WriteLine);`

This works because there is an overload of `Console.WriteLine` that takes an `int` argument and therefore conforms to the `IntAction` delegate type.

You can use `Filter` and `Act` to print only the even list elements (those divisible by 2) like this:

```
xs.Filter(delegate(int x) { return x%2==0; }).Act(Console.WriteLine);
```

Use anonymous methods to write an expression that prints only those list elements that are greater than or equal to 25.

An anonymous method may refer to local variables in the enclosing method. Use this fact and the `Act` method to compute the sum of an `IntList`'s elements (without writing any loops yourself).

`IntList.cs`

```
using System;
using System.Collections.Generic;

// Delegate types to describe predicates on ints and actions on ints.

public delegate bool IntPredicate(int x);
public delegate void IntAction(int x);

// Integer lists with Act and Filter operations.
// An IntList containing the element 7 9 13 may be constructed as
// new IntList(7, 9, 13) due to the params modifier.

class MyList<T> : List<T>
{
    public delegate bool IntPredicate2(T x);
    public delegate void IntAction2(T x);

    public MyList(params T[] elements) : base(elements) { }

    public void Act(IntAction2 f) {
        foreach (T i in this)
            f(i);
    }

    public MyList<T> Filter(IntPredicate2 p) {
        MyList<T> res = new MyList<T>();
        foreach (T i in this)
            if (p(i))
                res.Add(i);
        return res;
    }
}

class IntList : List<int> {

    public IntList(params int[] elements) : base(elements) { }
```

```
public void Act(IntAction f) {
    foreach (int i in this)
        f(i);
}

public IntList Filter(IntPredicate p) {
    IntList res = new IntList();
    foreach (int i in this)
        if (p(i))
            res.Add(i);
    return res;
}

class MyTest {
    public static void Main(String[] args) {
        // code here
        IntList list = new IntList(2, 23, 345, 40, 19);
        list.Act(Console.WriteLine);

        Console.WriteLine("Elements can be divided by 20");
        list.Filter(delegate(int x) { return x % 20==0; }).Act(Console.WriteLine);

        Console.WriteLine("Elements >= 25");
        list.Filter(delegate(int x) { return (x >= 20); }).Act(Console.WriteLine);

        // Compute total
        int total = 0;
        list.Act(delegate(int x) { total += x; });
        Console.WriteLine("Total: {0}", total);

        /* Extend for all type */
        Console.WriteLine("All double array");
        MyList<double> dList = new MyList<double>(3.34, 45.34, 40, 2.2);
        dList.Act(Console.WriteLine);

        Console.Read();
    }
}
```

Reference:

- 1) MSDN Document
- 2) <http://www.java2s.com/Tutorial/CSharp/CatalogCSharp.htm>
- 3) CD ROM C# Programming, Aptech Computer Education
- 4) [ebook] MSDN training, Introduction to C#, Microsoft Press