Agenda

- What are Generics?
- Implementation of Generic Classes
- Generic Interfaces
- Generic Methods
- Generics and Arrays
- Generic Delegates
- Defaults in Generics
- Constraining Types

What are Generics? (1 of 3)

- Generics let you tailor a method, class, structure, and interface to the precise data type.
- Generics introduces to the .NET Framework the concept of type parameters, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.
- By using Generic type parameter T you can write singles that other client code can use without incurring the cost or risk of runtime casts or boxing operations.
- Use Generic types to maximize code reuse, type safety, and performance.

What are Generics? (2 of 3)

- The most common use of Generics is to create collection classes.
- The .NET Framework class library contains several new Generic collection classes in the System.Collections.Generic namespace:
  - These should be used whenever possible in place of classes such as ArrayList in the System.Collections namespace.
- You can create your own Generic interfaces, classes, methods, events and delegates.
- Generic classes may be constrained to enable access to methods of particular data types.
- Information on the types used in a Generic data type may be obtained at run-time by means of reflection.
- Can overload Generic methods.

What are Generics? (3 of 3)

- Benefits of Generics:
  - Code reusability.
  - Faster code(no runtime costing).
  - World’s first cross-language Generics (not just for C#, but C++ and VB and other languages running on the CLR).

Generics Classes Implementation

```
// Declare the generic class
public class GenericList<T>
{
    void Add(T input) { }
}
class TestGenericList
{
    private class ExampleClass { }
    static void Main()
    {
        // Declare a list of type int
        GenericList<int> list1 = new GenericList<int>();

        GenericList<string> list2 = new GenericList<string>();

        // Declare a list of type ExampleClass
        GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
    }
}
```

Generic Class Declaration

Generic Type Parameter

Declaring a list of type int

Declaring a list of type string

Declaring a list of type ExampleClass

Generic Interfaces

- Interfaces can be defined for Generic collection classes, or for the Generic classes that represent items in the collection.
- The preference for Generic classes is to use Generic interfaces, such as IComparable<T> rather than IComparable in order to avoid boxing and unboxing operations on value types.
- The .NET Framework Class Library defines several Generic interfaces for use with the collection classes in the System.Collections.Generic namespace.
- Multiple interfaces can be specified as constraints on a single type, as follows:

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{ }
```

- An interface can define more than one type parameter, as follows:

```
interface IDictionary<K, V> { }
```

Generic Methods

- A Generic method is a method that is declared with type parameters:

```
static void Swap<T>(ref T lhs, ref T rhs)
 {
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
  }
```

Generics and Arrays

- This enables you to create Generic methods that can use the same code to iterate through arrays and other collection types. This technique is primarily useful for reading data in collections. The IList<T> interface cannot be used to add or remove elements from an array.

```
class Program
{
        static void Main()
         {
         int[] arr = { 0, 1, 2, 3, 4 };
           List<int> list = new List<int>();
                  ProcessItems<int>(arr);
         for (int x = 5; x < 10; x++) {
        list.Add(x); }
                  ProcessItems<int>(list);
 }
```

```
static void ProcessItems<T>(IList<T> coll)
{ // IsReadOnly returns True for the array and
False for the List. System.Console.WriteLine
("IsReadOnly returns {0} for this collection.",
coll.IsReadOnly);
 foreach (T item in coll)
{
 System.Console.Write(item.ToString() + " ");
}
 System.Console.WriteLine(); }
}
```

Generic Delegates

- A delegate can define its own type parameters. Code that references the Generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a Generic class or calling a Generic method.

```
namespace GenericDelegate
{
    public delegate void MyGenericDelegate<T>(T arg);
   class Program    {
      static void Main(string[] args)       {
            Console.WriteLine
              ("***** Generic Delegates *****\n");
            // Register target with 'traditional'
            //delegate syntax.
            MyGenericDelegate<string> strTarget =
            new MyGenericDelegate<string>(StringTarget);
            strTarget("Some string data");
            // Register target using method group conversion.
            MyGenericDelegate<int> intTarget = IntTarget;
            intTarget(9);
            Console.ReadLine();
}
```

```
Static void StringTarget(string arg)
        {
            Console.WriteLine("arg in uppercase is:
{0}", arg.ToUpper());
        }
        static void IntTarget(int arg)
        {
            Console.WriteLine("++arg is: {0}", ++arg);
        }
 }
}
```

Defaults in Generics

- In Generic classes and methods, one issue that arises is how to assign a default value to a parameterized type T when you do not know the following in advance:
  - Whether T will be a reference type or a value type.
  - If T is a value type, whether it will be a numeric value or a struct.
- Use the default keyword, which will return null for reference types and zero for numeric value types. For structs, it will return each member of the struct initialized to zero or null depending on whether they are value or reference types. For nullable value types, default returns a System.Nullable<T>, which is initialized like any struct.

```
public void ResetPoint()
{
    xpos = default(T);
    ypos = default(T);
}
```

Constraining Types

- By using constraining Types, it is possible to restrict the types that can be used to instantiate a Generic class.
- The possible constraints are:
  - Struct
  - Class
  - Base-class
  - Interface
- Class GenericClass<T> where T : constraint

```
{
}
```