Why?

Developers who are familiar with object-oriented programming know the benefts it offers. One of the big benefts that make developers extremely productive is code reuse, which is the ability to derive a class that inherits all of the capabilities of a base class. The derived class can simply override virtual methods or add some new methods to customize the behavior of the base class to meet the developer's needs. *Generics* is another mechanism offered by the common language runtime (CLR) and programming languages that provides one more form of code reuse: algorithm reuse

Basically, one developer defnes an algorithm such as sorting, searching, swapping, comparing, or converting. However, the developer defning the algorithm doesn't specify what data type(s) the algorithm operates on; the algorithm can be generically applied to objects of different types. Another developer can then use this existing algorithm as long as he or she indicates the specifc data type(s) the algorithm should operate on, for example, a sorting algorithm that operates on Int32s, Strings, etc., or a comparing algorithm that operates on DateTimes, Versions, etc.

Example

```
[Serializable]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
   IList, ICollection, IEnumerable {

   public List();
   public void Add(T item);
   public Int32 BinarySearch(T item);
   public void Clear();
   public Boolean Contains(T item);
   public Int32 IndexOf(T item);
   public Boolean Remove(T item);
   public void Sort();
   public void Sort(IComparer<T> comparer);
   public void Sort(Comparison<T> comparison);
   public T[] ToArray();

   public Int32 Count { get; }
   public T this[Int32 index] { get; set; }
}
```

The programmer who defned the generic List class indicates that it works with an unspecifed data type by placing the <T> immediately after the class name. When defning a generic type or method, any variables it specifes for types (such as T) are called *type parameters*. T is a variable name that can be used in source code anywhere a data type can be used. For example, in the List class defnition, you see T being used for method parameters (the Add method accepts a parameter of type T) and return types (the ToArray method returns a single-dimension array of type T)

WHERE?

Now that the generic List<T> type has been defned, other developers can use this generic algorithm by specifying the exact data type they would like the algorithm to operate on. When using a generic type or method, the specifed data types are referred to as *type arguments*. For example, a developer might want to work with the List algorithm by specifying a DateTime type argument

```
private static void SomeMethod() {
   // Construct a List that operates on DateTime objects
   List<DateTime> dtList = new List<DateTime>();

   // Add a DateTime object to the list
   dtList.Add(DateTime.Now);      // No boxing

   // Add another DateTime object to the list
   dtList.Add(DateTime.MinValue); // No boxing

   // Attempt to add a String object to the list
   dtList.Add("1/1/2004");        // Compile-time error

   // Extract a DateTime object out of the list
   DateTime dt = dtList[0];       // No cast required
}
```

Benefts

▪▪**Source code protection** The developer using a generic algorithm doesn't need to have access to the algorithm's source code.
▪▪**Type safety** When a generic algorithm is used with a specifc type, the compiler and the CLR understand this and ensure that only objects compatible with the specifed data type are used with the algorithm. Attempting to use an object of an incompatible type will result in either a compiler error or a run-time exception being thrown. In the example, attempting to pass a `String` object to the `Add` method results in the compiler issuing an error.
▪▪**Cleaner code** Because the compiler enforces type safety, fewer casts are required in your source code, meaning that your code is easier to write and maintain. In the last line of `SomeMethod`, a developer doesn't need to use a `(DateTime)` cast to put the result of the indexer (querying element at index 0) into the `dt` variable.
▪▪**Better performance** Before generics, the way to defne a generalized algorithm was to defne all of its members to work with the `Object` data type. If you wanted to use the algorithm with value type instances, the CLR had to box the value type instance prior to calling the members of the algorithm

Real Examples

1.Generic Repository interface

```
0 references
public interface IRepository<TEntity,TKey>
{
    0 references
    IEnumerable<TEntity> FindAll();
    0 references
    TEntity FindById(TKey key);
    0 references
    void Insert(TEntity entity);
    0 references
    void Update(TEntity update);
    0 references
    void Delete(TKey key);
}
```

2.Generic Proxy interface

```
0 references
public interface IProxy
{
    0 references
    bool CheckConn(string token);
    0 references
    string SendDataToServer(string token, string api, string data, Action success = null);
    0 references
    T GetDataFromServer<T>(string token, string api, Dictionary<string,string> parms = null)
       where T : class;
    0 references
    string GetDataFromServerValue(string token, string api);

}
```