Abstract
Generics is mechanism offered by the common language runtime (CLR) and programming languages that provides one more form of code reuse: algorithm reuse. Parametric polymorphism is the core principle behind generic programming languages and structures. It enables the creation of generic functions and data types that operate on values, regardless of data type. In C# 3.5 generics (a.k.a. Parametric Polymorphism) were added and used to refactor the code involving generalization, such as the collection classes. The initial versions of the collections classes are still available (System.Collections) for compatibility with prior code. But the generic versions of the collection classes (System.Collections.Generic) are now recommended for implementing collections in code that targeted .NET 3.0 or more recent releases. Besides its use in creating collections, generics also allow developers to create custom generic methods, classes, structs, interfaces, delegates, and events. Using generics can provide compile-time type safety, eliminate the performance penalty associated with boxing/unboxing, and simplify the code required to provide reusable solutions.

1. Introduction
Microsoft design guidelines that generic parameter variables should either are called T or least start with an uppercase T. The uppercase T stands for type, just as I stand for interface as in I Enumerable. Generics provide the following big benefits to developers:

- **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 specific type, the compiler and the CLR understand this and ensure that only objects compatible with the specified 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 runtime exception being thrown.

• **Cleaner Code:** The code is easier to maintain, since the compiler enforces type safety, fewer casts are required the code.

• **Better Performance:** Generic algorithm can be created to work with a specific value type and the CLR no longer has to do any boxing and casts are unnecessary. The CLR doesn’t have to check the type safety of the generated code and this enhances the performance of the algorithm.

```csharp
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
public static class MyApp {
    public static void Main() {
        ValueTypePerfTest();
        ReferenceTypePerfTest();
    }
    private static void ValueTypePerfTest() {
        const Int32 count = 10000000;
        using (new OperationTimer("List<Int32>")) {
            List l = new List(Count);
            for (Int32 n = 0; n < count; n++) {
                l.Add(n);
                Int32 x = l[n];
            }
            l = null; // Make sure this gets GC'd
        }
        using (new OperationTimer("ArrayList of Int32")) {
            ArrayList a = new ArrayList();
            for (Int32 n = 0; n < count; n++) {
                a.Add(n);
                Int32 x = (Int32) a[n];
            }
            a = null; // Make sure this gets GC'd
        }
    }
    private static void ReferenceTypePerfTest() {
        const Int32 count = 10000000;
        using (new OperationTimer("List<String>")) {
            List<String> l = new List<String>;
            for (Int32 n = 0; n < count; n++) {
                l.Add("X");
                String x = l[n];
            }
            l = null; // Make sure this gets GC'd
        }
    }
}
```
l = null; // Make sure this gets GC’d
}
using (new OperationTimer("ArrayList of String")) {
ArrayList a = new ArrayList();
for (Int32 n = 0; n < count; n++) {
a.Add("X");
String x = (String)a[n];
}
a = null; // Make sure this gets GC’d
}
}

// This class is useful for doing operations performance timing
internal sealed class OperationTimer : IDisposable {
private Int64 m_startTime;
private String m_text;
private Int32 m_collectionCount;
public OperationTimer(String text) {
PrepareForOperation();
m_text = text;
m_collectionCount = GC.CollectionCount(0);
// This should be the last statement in this
// method to keep timing as accurate as possible
m_startTime = Stopwatch.GetTimestamp();
}
public void Dispose() {
Console.WriteLine("(0,6:###.00) seconds (GCs={1,3}) {2}",
(StopWatch.GetTimestamp() – m_startTime) /
(Double) StopWatch.Frequency, GC.CollectionCount(0) –
m_collectionCount, m_text);
}
private static void PrepareForOperation() {
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
}

When I run this program
.20 seconds (GCs = 0) List<Int32>
3.30 seconds (GCs = 45) ArrayList of Int32
.50 seconds (GCs = 6) List<String>
.58 seconds (GCs = 6) ArrayList of String

The output here shows that using the generic List with Int32 type is much faster than using the non-generic ArrayList algorithm with Int32. Also using the value type – Int32 with ArrayList causes a lot of boxing operations to occur which results in 45 GC whereas List algorithm requires 0 GC.
So it doesn’t appear that the generic List algorithm is of any benefit here. But it gives a cleaner code and compiles time type safety.

- **Generics inside FCL**
Microsoft recommends that developers use the generic collection classes and now discourages use of the non-generic collection classes for several reasons. First, the non-generic collection classes are not generic and so you don’t get the type safety, cleaner code, and better performance that you get when you use generic collection classes. Second the generic classes have a better object model than non-generic classes. For example, fewer methods are virtual, resulting in better performance, and new members have been added to the generic collections to provide new functionality.
The FCL ships with many generic interface definitions so that the benefits of generics can be realized when working with interface as well. The commonly used interfaces are contained in the `System.Collections.Generic` namespace.

2. **Infrastructure for Generics**
Microsoft had to provide the following for Generics to work properly.

- Create new IL instructions that are aware of type arguments
- Modify the format of existing metadata tables so that type names and methods with generic parameters could be expressed.
- Modify the various programming languages to support the new syntax, allowing developers to define and reference generic types and methods
- Modify the compilers to emit the new IL instructions and the modified metadata format.
- Modify the just-in-time(JIT) compiler to process the new type argument aware IL instructions that produce the correct native code.
- Create new reflection members so that developers can query types and members to determine if they have generic parameters. Created new methods using Reflection had to be defined so that developers could create generic type and method definitions at runtime.
- Modify the debugger to show and manipulate generic types, members, fields and local variables.
- Modify the Microsoft VS Intellisense feature to show specific member prototypes when using a generic type or a method with a specific data type.

1. **Open and Closed Types**
The CLR creates an internal data structure for each and every type in use by an application. These data structures are called type objects. A type with generic type parameters is still considered a type and the CLR will create an internal type object for each of these. This applies to reference types, value types, interface types and delegate types. A type with generic type parameters is called Open type and the CLR doesn’t allow any instance of an open type to be constructed.

When code references a generic type it can specify a set of generic type arguments. If actual data types are passed in for all of the type arguments, the type is called a closed type, and the CLR does allow instances of a closed type to be constructed.

For e.g.
```csharp
using System;
using System.Collections;
using System.Collections.Generic;
// A partially specified open type
```
internal sealed class DictionaryStringKey<TValue> : Dictionary<String, TValue> {
}
public static class MyApp {
    public static void Main() {
        Object o = null;
        //Dictionary<,,> is an open type having 2 new type parameters
        Type t = typeof(Dictionary<,,>);
        // try to create an instance of this type (fails)
        o = CreateInstance(t);
        Console.WriteLine();
        //DictionaryStringKey<,,> is an open type having 1 type parameter
        t = typeof(DictionaryStringKey<>);
        // Try to create an instance of this type fails
        o = CreateInstance(t)
        Console.WriteLine();
        // DictionaryStringKey<Guid> is a closed type
        t = typeof(DictionaryStringKey<Guid>);
        // Try to create an instance of this type succeeds
        o = CreateInstance(t)
        Console.WriteLine(“Object types=”+o.GetType());
    }
    private static Object CreateInstance(Type t) {
        Object o = null;
        try {
            o = Activator.CreateInstance(t);
            Console.Write(“Created instance of (0)”,t.ToString());
        }
        catch (ArgumentException e) {
            Console.WriteLine(e.Message);
        }
        return o;
    }
}

When we execute this code, we get the following output:
Cannot create an instance of DictionaryStringKey 1[TValue] because Type.ContainsGenericParameters is true.
Created an instance of DictionaryStringKey `1[System.Guid]
Object Type = DictionaryStringKey `1[System.Guid]

In the output we see that the names end with a backtick (`) followed by a number, it is type’s arity which indicates the number of type parameters required by the type.
3. Generic Types And Inheritance

A generic type is a type, and it can be derived from any other type. When you use generic type and specify type arguments, you are defining a new type object in the CLR, and the new type object is derived from whatever type generic type was derived from. I.e. List<T> is derived from Object, List<String> and List<Guid> are also derived from Object. Similarly, Dictionary<StringKey<TValue> is derived from Dictionary<StringKey<String, TValue>, Dictionary<StringKey<Guid> is also derived from Dictionary<String, Guid>. Consider an example below:

```csharp
internal class Node {
    protected Node m_next;
    public Node(Node next) {
        m_next = next;
    }
}
internal sealed class TypedNode<T> : Node {
    public T m_data;
    public TypedNode(T data) : this(data, null) {
    }
    public TypedNode(T data, Node next) : base(next) {
        m_data = data;
    }
    public override String ToString() {
        return m_data.ToString() = ((m_next != null) ? m_next.ToString() : String.Empty);
    }
}
```

Now the main code will be as follows:

```csharp
internal class Node {
    protected Node m_next;
    public Node(Node next) {
        m_next = next;
    }
}
internal sealed class TypedNode<T> : Node {
    public T m_data;
    public TypedNode(T data) : this(data, null) {
    }
    public TypedNode(T data, Node next) : base(next) {
        m_data = data;
    }
    public override String ToString() {
        return m_data.ToString() = ((m_next != null) ? m_next.ToString() : String.Empty);
    }
}
```

Now the main code will be as follows:

```csharp
private static void DifferentDataLinkedList() {
    Node head = new TypedNode<Char>(",");
    head = new TypedNode<DateTime>(DateTime.Now, head);
    head = new TypedNode<String>("Today is", head);
    Console.WriteLine(head.ToString());
}
```

I. Generic Type Identity

C# does offer a way to use simplified syntax to refer to a generic closed type while not affecting type equivalence at all; you can use the good old using directive at the top of your source code file. Here is an example:

```csharp
using DateTimeList = System.Collections.Generic.List<System.DateTime
```

Using directive is really just defining a symbol called DateTimeList. As the code compiles, the compiler substitutes all occurrences of DateTimeList with System.Collections.Generic.List<System.DateTimeThis just allows developers to use a simplified syntax without affecting the actual meaning of the code, and therefore, type identity and equivalence are maintained. So now when the following line executes the sameType will initialized to true.
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));

II. Code Explosion
When a method that uses generic type parameters is JIT-compiled, the CLR takes the method IL, substitutes the specified type arguments, and then creates native code that is specific to that method operating on the specified data types. The CLR keeps generating the native code for every method/type combination. This is referred to as code explosion.

Fortunately, CLR has some optimizations built into it to reduce code explosion. First, if a method is called for a particular type argument, and later the method is called again using the same type argument, the CLR will compile the code for this method/type combination just once. So if assembly uses List<DateTime>, and a completely different assembly also uses List<DateTime>, the CLR will compile the methods for List<DateTime>. This reduces the code explosion. The CLR has another optimization: the CLR considers all reference type arguments to be identical and the code can be shared. For example, the code compiled by the CLR for List<String>’s method can be used for List<Stream>’s methods, since String and Stream are both reference types. In fact, for any reference type, the same code will be used. But if the type argument is a value type, the CLR must produce native code specifically for that value type. The reason is the value types can vary in size.

III. Generic Interfaces
CLR supports Generic Interface to avoid boxing and loss of compile time type safety. A reference or value type can implement a generic interface by specifying type arguments, or a type can implement a generic interface by leaving the type arguments unspecified.

The definition of a generic interface in the System.Collections.Generic namespace that is part of FCL:
public interface IEnumerator<T> : IDisposable, IEnumerator {
    T Current { get; }  
}

Eg of Generic Interface
internal sealed class Triangle : IEnumerator<Point> {
    private Point[] m_vertices;
    // IEnumerator<Point>’s Current property is of type Point
    public Point Current { get {……} }
    …..
}

Now the Generic class that implements Generic Interface
internal sealed class ArrayEnumerator<T> : IEnumerator<T> {
    private T[] m_array;
    // IEnumerator<T>’s Current property is of type T
    public T Current { get {……} }
    ….
}

IV. Generic Delegates
The CLR supports generic delegates to ensure that any type of object can be passed to a callback method in a type-safe way. Furthermore generic delegates allow a value type instance to be passed to
a callback method without any boxing. “Delegates,” a delegate is really just a class definition with four methods: a constructor, an invoke method, a BeginInvoke method, and an EndInvoke method. When you define a delegate type that specifies type parameters, the compilers emits the delegate class’s methods and the type parameters are applied to any methods having parameters/return values of the specified type parameter.

For example, if you define a generic delegate like this:

```csharp
public delegate TReturn CallMe<TReturn, TKey, TValue>(TKey key, TValue value);
```

The compiler turns that into a class that logically looks like this:

```csharp
public sealed class CallMe<TReturn, TKey, TValue> : MulticastDelegate {
    public CallMe(Object object, IntPtr method);
    public virtual TReturn Invoke(TKey key, TValue value);
    public virtual TReturn IAsyncResult BeginInvoke(TKey key, TValue value,
                                                  AsyncCallback callback, Object object);
    public virtual TReturn EndInvoke(IAsyncResult result);
}
```

It is recommended that one should use the generic Action and Func delegates that come predefined in the Framework Class Library wherever possible.


Each of a delegate’s generic type parameters can be cast to a variable of generic delegate type where the generic parameter type differs, A generic type parameter can be of the following:

- Invariant : A generic type parameter that cannot be changed.
- Contravariant : A generic type parameter that can change from a class to a class derived from it. The contravariant can appear as only input parameters with in keyword
- Covariant : A generic type parameter that can change from a class to one of its base classes. In C#, you indicate covariant generic type parameters with the out keyword which can appear only in output positions such as a method’s return type.

```csharp
public delegate TResult Func<in T, out TResult>(T arg);
```

In this generic type parameter T is marked with the in keyword making it contravariant; and the generic type parameter TResult is marked with the out keyword, making it covariant.

If I have variable declared as follows:

```csharp
Func<Object, ArgumentException> fn1 = null;
I can cast it to another Func type, where the generic type parameters are different:
Func<String, Exception> fn2 = fn1; // no explicit cast is required here
Exception e = fn2(“ “);
```

Here fn1 refers to a function that accepts an Object and returns an Argument Exception. The fn2 variable wants to refer to a method that takes a String and returns an Exception. Since you can pass a String to a method that wants an Object, and since you can take the result of a method that returns an Argument Exception and treat it as an Exception, the code above compiler and is known at compile time to preserve type safety.
Note: Variance is not possible for value types because boxing would be required. Also variance is not allowed on a generic type parameter if an argument of that type is passed to a method using the out or ref keyword. And CLR would throw a following exception if it find this kind of statement:
Invalid variance: They type parameter ‘T’ must be invariantly valid on ‘SomeDelegate<T>.Invoke(ref T)’. ‘T’ is contravariant.”
delegate void SomeDelegate<in T>(ref T t);

When using delegates that take generic arguments and return values, it is recommended to always specify the in and out keywords for contravariance and covariance whenever possible as doing this has no ill effects and enables your delegate to be used in more scenarios. Here is an example of an interface with a contravariant generic type parameter:

public interface IEnumerator<out T> : IEnumerator {
    Boolean MoveNext();
    T Current { get; }
}
Since T is contravariant, it is possible to have the following code compile and run successfully:

// This method accepts an IEnumerable of my reference type
Int32 Count(IEnumerable<Object> collection) {…}

…. //The call below passes an IEnumerable<String> to count
Int32 c = count(new[]{ "Grant" });
for this reason the compiler team forces you to be explicit when declaring a generic type parameter. Then if you attempt to use this type parameter in a context that doesn’t match how you declared it, the compiler issues an error letting you know that you are attempting to break the contract. If you then decide to break the contract by adding in or out on generic type parameters, you should expect to have to modify some of the code sites that were using the out contract.

I. Generic Methods
When you define a generic class, struct, or interface, any methods defined in these types can refer to a type parameter specified by the type. A type parameter can be used as a method’s parameter, a method’s return value, or as a local variable defined inside the method. However, the CLR also supports the ability for a method to specify its very own type parameters. And these type parameters can also be used for parameters, return values, or local variables.

internal sealed class GenericType<T> {
    private T m_value;
    public GenericType(T value) { m_value = value; }
    public TOutput Converter<TOutput>() {
        TOutput result = (TOutput) Convert.ChangeType(m_value, typeof(TOutput));
        return result;
    }
}

In this example, you can see that the GenericType class defines its own type parameter(T), and the Converter method defines its own type parameter(TOutput). This allows a GenericType to be constructed to work with any type. The converter method can convert the object referred to by the m_value field to various types depending on what type argument is passed to it when called. The ability to have type parameters and method parameters allows for phenomenal flexibility.

A reasonably good example of a generic method is the two method
private static void swap<T>(ref T o1, ref T o2) {
Soumya B.S., Jayaprakash S.T.: Templating and Generics in .NET Framework 4.0

II. Generic Methods And Type Inference

To help improve code creation, readability, and maintainability, the C# compiler offers type inference when calling a generic method. Type inference means that the compiler attempts to determine the type to use automatically when calling a generic method.

Here is some code that demonstrates type inference:

```csharp
private static void CallingSwapUsingInference() {
    Int32 n1 = 1, n2 = 2;
    Swap(ref n1, ref n2);// Calls Swap<Int32>
    String s1 = “Aidan”;
    Object s2 = “Grant”;
    Swap(ref s1, ref s2);// Error, type can’t be inferred
}
```

In this code, first call to Swap compiler infers n1 and n2 are Int32 and hence it will invoke Swap with Int32 type parameter. In the second call compiler sees that s1 is String and s2 is Object. Since s1 and s2 are variables of different data types, the compiler can’t accurately infer the type to use for swap’s type argument, and it issues invalid type arguments error for method ‘Swap<T>(ref T, ref T)’.

Another type is a type that can be defined with multiple methods with one of its methods taking a specific data type and another taking a generic type parameter as shown below in the code:

```csharp
private static void Display(String s) {
    Console.WriteLine(s);
}
private static void Display<T>(T o) {
    Display(o.ToString()); //Calls Display(String)
}
```

Here are some ways to call the Display method
Display(“Jeff”);  // Calls Display(String)
Display(123);  // Calls Display<T>(T)
Display<String>(“Adrian”);  // Calls Display<T>(T)

The C# compiler always prefers a more explicit match over a generic match, and therefore, it generates a call to the non-generic Display method that takes a String. For the second call, the compiler can’t call the non-generic Display method that takes a String, so it must call the generic Display method. By the way, it is fortunate that the compiler always prefers the more explicit match; if the compiler had preferred the generic method, because the generic display method calls Display again there would have been infinite recursion.

5. Verifiability And Constraints

A constraint is a way to limit the number of types that can be specified for a generic argument. Limiting the number of types allows you to do more with those types. Here is a new version of the Min method that specifies a constraint:

```csharp
public static T Min<T> o1, o2> where T : IComparable<T> {
    if (o1.CompareTo(o2) < 0) return o1;
    return o2;
}
```

The C# where token tells the compiler that any type specified for T must implement the generic IComparable interface of the same type(T). Because of this constraint, the compiler now allows the method to call the CompareTo method since this method is defined by the IComparable<T> interface.

Now when code references a generic type or method, the compiler is responsible for ensuring that a type argument that meets the constraints is specified.

For e.g.

```csharp
private static void CallMin() {
    Object o1 = “Jeff”, o2 = “Richter”;
    Object oMin = Min<Object>(o1, o2); // Error
}
```

The compiler issues the error because system.Object doesn’t implement the IComparable<Object> interface. In fact, system.Object doesn’t implement any interfaces at all.

The CLR doesn’t allow overloading based on the type parameter names or constraints you can overload types or methods based only on arity. The following e.g. shows that

```csharp
// It is OK to define the following types
internal sealed class AType {}
internal sealed class AType <T>{}
internal sealed class AType <T1, T2>{}
//error : conflicts with AType<T> that has no constraints

internal sealed class AType<T> where T: IComparable<T> {}

//Error: conflicts with AType<T1, T2>
internal sealed class AType <T1, T2>{}
internal sealed class AnotherType {
    private static void M() {}
    private static void M<T>() {}
}
private static void M<T1, T2>() {}

//Error: conflicts with M<T> that has no constraints
private static void M<T>() where T : IComparable<T> {}

//Error
private static void M<T3, T4>() {}

In fact, the overriding method is not allowed to specify any constraints on its type parameters at all. However, it can change the names of the type parameters. Similarly, when implementing an interface method, the method must specify the same number of type parameters as the interface method and these type parameters will inherit the constraints specified on them by the interface’s method.

E.g.
internal class Base {
    public virtual void M<T1, T2>()
        where T1 : struct
    where T2 : class {
    }
}

internal sealed class Derived : Base {
    public override void M<T3, T4>()
        where T3 : EventArgs //Error
        where T4 : class //Error
    {
    }
}

Notice that you can change the names of the type parameters as in the example from T1 to T3 and T2 to T4; however, you cannot change constraints.

A type parameter can be constrained using a primary constraint, a secondary constraint, and constructor constraint.

I. Primary Constraint

A primary constraint can be reference type that identifies a class that is not sealed. You cannot specify one of the following special reference types: System.Object, System.Array, System.Delegate, System.MulticastDelegate, System.ValueType, System.EventArgs or System.Void.

When specifying a reference type constraint, you are promising the compiler that a specified type argument will either be of the same type or of a type derived from the constant type. For e.g.

internal sealed class PrimaryConstraintOfStream<T> where T : Stream {
    public void M(T stream) {
        stream.Close(); //OK
    }
};

In this class definition the type parameter T has a primary constraint of Stream. This tells the compiler that code using PrimaryConstraintOfStream must specify a type argument of Stream or a type derived from Stream. If a type parameter doesn’t specify a primary constraint, System.Object is assumed. However, the C# compiler issues an error message if you explicitly specify System.Object in your source code.
There are two special primary constraints: class and struct. The class constraint promises the compiler that a specified type argument will be reference type. Any class type, interface type delegate type or array type satisfies this constraint. For e.g.

```csharp
internal sealed class PrimaryConstraintOfClass<T> where T : class {
    public void M() {
        T temp = null; // Allowed because T must be a reference type
    }
}
```

In this example setting `temp` to `null` is legal because `T` is known to be a reference type, and all reference type variables can be set to `null`. If `T` were unconstrained, the code above would not compile because `T` could be a value type, and value type variables cannot be set to `null`.

The struct constraint promises the compiler that a specified type argument will be a value type. Any value type, including enumerations, satisfies this constraint. However, the compiler and the CLR treat any System.Nullable<T> value type as a special type and nullable types do not satisfy this constraint. The reason is because the Nullable<T> type constrains its type parameter to struct, and the CLR wants to prohibit a recursive type such as Nullable<Nullable<T>>

```csharp
internal sealed class PrimaryConstraintOfStruct<T> where T : struct {
    public static T Factory() {
        // Allowed because all value types implicitly have a public parameterless constructor
        return new T();
    }
}
```

In this example, newing up a `T` is legal because `T` is known to be a value type and all value types implicitly have a public, parameterless constructor. If `T` were unconstrained, constrained to a reference type or constrained to class, the above code would not compile because some reference types do not have public, parameterless constructors.

### II. Secondary Constraint

A type parameter can specify zero or more secondary constraints where a secondary constraint represents an interface type. When specifying an interface type constraint, you are promising the compiler that a specified type argument will be a type argument that must specify a type that implements all of the interface constraints.

There is another kind of secondary constraint called a type parameter constraint. This kind of constraint is used much less often than interface constraint. It allows a generic type of method to indicate that there must be a relationship between specified type arguments. A type parameter can have zero or more type constraints applied to it. Here is a generic method that demonstrates the use of a type parameter constraint:

```csharp
private static List<TBase> ConvertIList<T, TBase>(IList<T> list) where T : TBase {
    List<TBase> baseList = new List<TBase>(list.Count);
    for (Int32 index = 0; index < list.Count; index++) {
        baseList.Add(list[index]);
    }
    return baseList;
}
```

The `ConvertIList` method specifies two types parameters in which the `T` parameter is constrained by the `TBase` type parameter. This means that whatever type argument is specified for `T`, the type argument must be compatible with whatever type arguments is specified for `TBase`. Here is a method showing some legal and illegal calls to `ConvertIList`:
private static void CallingConvertIList(){
    //Construct and initialize a List<String> (which implements IList<String>)
    IList<String> ls = new List<String>();
    ls.Add("A String");
    //Convert the IList<String> to an IList<Object>
    IList<Object> lo = ConvertIList<String, Object>(ls);
    //Convert the IList<String> to an IList<IComparable>
    IList<IComparable> lc = ConvertIList<String, IComparable>(ls);
    //Convert the IList<String> to an IList<IComparable<String>>
    IList<IComparable<String>> lcs = ConvertIList<String, IComparable<String>>(ls);
    //Convert the IList<String> to an IList<String>
    IList<String> ls2 = ConvertIList<String, String>(ls);
    //Convert the IList<String> to an IList<Exception>
    IList<Exception> le = ConvertIList<String, Exception>(ls);  //Error

    In the first call to ConvertIList, the compiler ensures that String is compatible with Object. Since
    String is derived from Object, the first call adheres to the type parameter constraint. In the second call
    to ConvertIList, the compiler ensures that String is compatible with IComparable. Since String
    implements the IComparable interface, the second call adheres to the type parameter constraint. In the
    third call to ConvertIList, the compiler ensures that String is compatible with IComparable<String>
    Since String implements the IComparable<String> interface, the third call adheres to the type
    parameter constraint. In the fourth call to ConvertIList, the compiler knows that String is compatible
    with itself. In the fifth call to ConvertIList, the compiler ensures that string is compatible with
    Exception. Since String is not compatible with Exception, the fifth call doesn’t adhere to they type
    parameter constraint, and the compiler issues the following message: “error CS0311: The type string
    cannot be used as type parameter ‘T’ in the generic type or method
    ‘Program.ConvertIList<T,TBase>(System.Collections.Generic.IList<T. There is no implicit reference
    conversion from string to System.Exception”.

    III. Constructor Constraints

    A type parameter can specify zero constructor constraints or one constructor constraint. When
    specifying a constructor constraint, you are promising the compiler that a specified type argument
    will be a non-abstract type that implements a public, parameterless constructor. Note that the C#
    compiler considers it an error to specify a constructor constraint with the struct constraint because it is
    redundant; all value types implicitly offer a public, parameterless constructor.
    e.g.
    internal sealed class ConstructorConstraint<T> where T : new() { public static T Factory() {
    // Allowed because all value types implicitly
    // have a public, parameterless constructor and //because the constraint requires that any //specified
    reference type also have a public, //parameterless constructor
    return new T();
    }
    }

    In the above example, newing up a T is legal because T is known to be a type that has a public,
    parameterless constructor. This is certainly true of all value types, and the constructor constraint
    requires that it be true of any reference type specified as a type argument.
Casting Generic Type:

Casting a generic type variable to another type is illegal unless you are casting to a type compatible with a constraint:

```csharp
private static void CastingGenericTypeVariable1<T>(T obj) {
    Int32 x = (Int32) obj; //Error
    String s = (String) obj; //Error
}
```

The compiler issues an error on both lines above because T could be any type, and there is no guarantee that the casts will succeed. You can modify this code to get it to compile by casting to `Object` first:

```csharp
private static void CastingAGenericTypeVariable2<T>(T obj) {
    Int32 x = (Int32) (object) obj; // No Error
    String s = (String) (Object) obj; // No Error
}
```

If a casting of a reference type needs to be done we can use ‘as’ operator. For e.g.

```csharp
private static void CastingAGenericTypeVariable3<T>(T obj) {
    String s = obj as String; // No error
}
```

Default value for Generic Type Variable:

Setting a generic type variable to null is illegal unless the generic type is constrained to a reference type.

```csharp
private static void SettingAGenericTypeVariableToNull<T>() {
    T temp = null; //C50403 – Cannot convert null to type parameter T
}
```

Since T is unconstrained, it could be a value type, and setting a variable of a value type to null is not possible. If T were constrained to a reference type, setting temp to null would compile and run just fine. C# team felt that it would be useful to give developers the ability to set a variable to a default value. So the C# compiler allows you to use the default keyword to accomplish this:

```csharp
private static void SettingAGenericTypeVariableToDefaultValue<T>(){
    T temp = default(T); // OK
}
```

The use of the default keyword above tells the C# compiler and the CLR’s JIT compiler to produce code to set temp to null if T is a reference type and to set temp to all-bits-zero if T is a value type.

IV. Comparison Of Generic Type Variables

Comparing a generic type variable to null by using the == or != operator is legal regardless of whether the generic type is constrained:

```csharp
private static void ComparingAGenericTypeVariableWithNull<T>(T obj) {
    if(obj == null) { /* Never executes for value type */ }
}
```

Since T is unconstrained, it could be a reference type or a value type. If T is a value type, obj can never be null. The C# compiler does not issue an error, instead, it compiles the code just fine. When this method is called using a type argument that is a value type, the JIT compiler sees that the if statement can never be true, and the JIT compiler will not emit the native code for the if test or the code in the braces. If I had used the != operator, the JIT compiler would not emit the code for the if test and it will emit the code inside the if’s braces.

By the way, if T had been constrained to a struct, the compiler would have thrown an error.

Comparing two Generic Type variables.
Comparing two variables of the same generic type is illegal if the generic type parameter is not known to be a reference type:

private static void ComparingTwoGenericTypeVariables<T>(T o1, T o2) {
    if(o1 == o2) {} //Error
}

In this example T is unconstrained, and whereas it is legal to compare two reference type variables with one another, it is not legal to compare two value type variables with one another unless the value type overloads the == operator.

If T had been constrained to a struct, the compiler would have thrown an error.

Avoid Generic Type as Operands

The operators such as +, –, *, and / can’t be applied to variables of a generic type because the compiler doesn’t know the type at compile time. This means that you can’t use any of these operators with variables of a generic type. So it is impossible to write a mathematical algorithm that works on an arbitrary numeric data type.

The real takeaway here is that by choosing C# generics we are not using just a feature of the compiler, but also a feature of the runtime. A generic type such as List<T> maintains its generic-ness (genericty) after it has been compiled. Or, to look at it another way, the substitution that the C++ compiler does at compile time is done at JIT time in the C# generic world. This illustrates that generics in C# are not like templates in C++. Thus Generics are used to write code that is reusable across different types. Generics can be used with methods, and the following types: class, structure, interface, and delegate. Generic code contains a type parameter (<T>) which specifies the data type which will be processed. Constraints can be added to type parameters to restrict the kinds of types which can be used for arguments.

6. Conclusion

C# generics are an invaluable part of your development arsenal. They improve performance, type safety and quality, reduce repetitive programming tasks, simplify the overall programming model, and do so with elegant, readable syntax. Generics are just one of the tools in your belt; make sure to learn how to use it, and more importantly, when its use is appropriate. No doubt future versions of C# will continue to evolve generics, adding new capabilities and extending generics to other areas of .NET Framework.

Acknowledgment

To my son Jaishnav without whose never-failing helpfulness and encouragement this article would have been finished in half the time. Dedicated to India, whatever that is.

References