CHAPTER 12

Generics

In this chapter:

Generics in the Framework Class Library 270
Generics Infrastructure 271
Generic Interfaces 277
Generic Delegates 278
Delegate and Interface Contra-variant and Covariant Generic Type Arguments 279
Generic Methods 281
Generics and Other Members 284
Verifiability and Constraints 284

Developers who are familiar with object-oriented programming know the benefits it offers. One of the big benefits 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 defines an algorithm such as sorting, searching, swapping, comparing, or converting. However, the developer defining 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 specific data type(s) the algorithm should operate on, for example, a sorting algorithm that operates on `Int32`s, `String`s, etc., or a comparing algorithm that operates on `DateTime`s, `Version`s, etc.

Most algorithms are encapsulated in a type, and the CLR allows the creation of generic reference types as well as generic value types, but it does not allow the creation of generic enumerated types. In addition, the CLR allows the creation of generic interfaces and generic delegates. Occasionally, a single method can encapsulate a useful algorithm, and therefore, the CLR allows the creation of generic methods that are defined in a reference type, value type, or interface.

Let's look at a quick example. The Framework Class Library (FCL) defines a generic list algorithm that knows how to manage a set of objects; the data type of these objects is not specified by the generic algorithm. Someone wanting to use the generic list algorithm can specify the exact data type to use with it later.

The FCL class that encapsulates the generic list algorithm is called `List<T>` (pronounced List of Tee), and this class is defined in the `System.Collections.Generic` namespace. Here is what this class definition looks like (the code is severely abbreviated).

```
[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 defined the generic `List` class indicates that it works with an unspecified data type by placing the <T> immediately after the class name. When defining a generic type or method, any variables it specifies 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 definition, 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). Another example is the indexer method (called `this` in C#). The indexer has a `get` accessor method that returns a value of type T and a `set` accessor method that accepts a parameter of type T. Because the T variable can be used anywhere that a data type can be specified, it is also possible to use T when defining local variables inside a method or when defining fields inside a type.

Note Microsoft's design guidelines state that generic parameter variables should either be called T or at least start with an uppercase T (as in `TKey` and `TValue`). The uppercase T stands for *type*, just as an uppercase `I` stands for *interface* (as in `IComparable`).

Now that the generic `List<T>` type has been defined, 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 specified 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.

Here is some code that shows this.

```
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
}
```

Generics provide the following big benefits to developers as exhibited by the code just shown:

- **Source code protection** The developer using a generic algorithm doesn't need to have access to the algorithm's source code. With C++ templates, however, the algorithm's source code must be available to the developer who is using the algorithm.
- **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 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 define a generalized algorithm was to define 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. As discussed in Chapter 5, "Primitive, Reference, and Value Types," boxing causes memory allocations on the managed heap, which causes more frequent garbage collections, which, in turn, hurt an application's performance. Because a generic algorithm can now be created to work with a specific value type, the instances of the value type can be passed by value, and the CLR no longer has to do any boxing. In addition, because casts are not necessary (see the previous bullet), the CLR doesn't have to check the type safety of the attempted cast, and this results in faster code too.

To drive home the performance benefits of generics, I wrote a program that tests the performance of the generic `List` algorithm against the FCL's non-generic `ArrayList` algorithm. In fact, I tested the performance of these two algorithms by using both value type objects and reference type objects. Here is the program itself.

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

public static class Program {
   public static void Main() {
      ValueTypePerfTest();
      ReferenceTypePerfTest();
   }

   private static void ValueTypePerfTest() {
      const Int32 count = 100000000;

      using (new OperationTimer("List<Int32>")) {
         List<Int32> l = new List<Int32>();
         for (Int32 n = 0; n < count; n++) {
            l.Add(n);                // No boxing
            Int32 x = l[n];          // No unboxing
         }
         l = null;  // Make sure this gets garbage collected
      }

      using (new OperationTimer("ArrayList of Int32")) {
         ArrayList a = new ArrayList();
         for (Int32 n = 0; n < count; n++) {
            a.Add(n);                  // Boxing
            Int32 x = (Int32) a[n];    // Unboxing
         }
         a = null;  // Make sure this gets garbage collected
      }
   }

   private static void ReferenceTypePerfTest() {
      const Int32 count = 100000000;

      using (new OperationTimer("List<String>")) {
         List<String> l = new List<String>();
         for (Int32 n = 0; n < count; n++) {
            l.Add("X");                // Reference copy
            String x = l[n];           // Reference copy
         }
         l = null;  // Make sure this gets garbage collected
      }

      using (new OperationTimer("ArrayList of String")) {
         ArrayList a = new ArrayList();
         for (Int32 n = 0; n < count; n++) {
            a.Add("X");                // Reference copy
            String x = (String) a[n];  // Cast check & reference copy
```

```
         }
         a = null;  // Make sure this gets garbage collected
      }
   }
}

// This class is useful for doing operation performance timing
internal sealed class OperationTimer : IDisposable {
   private Stopwatch m_stopwatch;
   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_stopwatch = Stopwatch.StartNew();
   }

   public void Dispose() {
      Console.WriteLine("{0} (GCs={1,3}) {2}", (m_stopwatch.Elapsed),
         GC.CollectionCount(0) - m_collectionCount, m_text);
   }

   private static void PrepareForOperation() {
      GC.Collect();
      GC.WaitForPendingFinalizers();
      GC.Collect();
   }
}
```

When I compile and run a release build (with optimizations turned on) of this program on my computer, I get the following output.

```
00:00:01.6246959 (GCs=  6) List<Int32>
00:00:10.8555008 (GCs=390) ArrayList of Int32
00:00:02.5427847 (GCs=  4) List<String>
00:00:02.7944831 (GCs=  7) ArrayList of String
```

The output here shows that using the generic `List` algorithm with the `Int32` type is much faster than using the non-generic `ArrayList` algorithm with `Int32`. In fact, the difference is phenomenal: 1.6 seconds versus almost 11 seconds. That's ~7 times faster! In addition, using a value type (`Int32`) with `ArrayList` causes a lot of boxing operations to occur, which results in 390 garbage collections. Meanwhile, the `List` algorithm required 6 garbage collections.

The result of the test using reference types is not as momentous. Here we see that the times and number of garbage collections are about the same. So it doesn't appear that the generic `List` algorithm is of any benefit here. However, keep in mind that when using a generic algorithm, you also get cleaner code and compile-time type safety. So although the performance improvement is not huge, the other benefits you get when using a generic algorithm are usually an improvement.

Note You do need to realize that the CLR generates native code for each method the first time the method is called for a particular data type. This will increase an application's working set size, which will hurt performance. I will talk about this more in the "Generics Infrastructure" section of this chapter.

Generics in the Framework Class Library

Certainly, the most obvious use of generics is with collection classes, and the FCL defines several generic collection classes available for your use. Most of these classes can be found in the `System.Collections.Generic` namespace and the `System.Collections.ObjectModel` namespace. There are also thread-safe generic collection classes available in the `System.Collections.Concurrent` namespace. Microsoft recommends that programmers 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 the 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 collection classes implement many interfaces, and the objects that you place into the collections can implement interfaces that the collection classes use for operations such as sorting and searching. The FCL ships with many generic interface definitions so that the benefits of generics can be realized when working with interfaces as well. The commonly used interfaces are contained in the `System.Collections.Generic` namespace.

The new generic interfaces are not a replacement for the old non-generic interfaces; in many scenarios, you will have to use both. The reason is backward compatibility. For example, if the `List<T>` class implemented only the `IList<T>` interface, no code could consider a `List<DateTime>` object an `IList`.

I should also point out that the `System.Array` class, the base class of all array types, offers many static generic methods, such as `AsReadOnly`, `BinarySearch`, `ConvertAll`, `Exists`, `Find`, `FindAll`, `FindIndex`, `FindLast`, `FindLastIndex`, `ForEach`, `IndexOf`, `LastIndexOf`, `Resize`, `Sort`, and `TrueForAll`. Here are examples showing what some of these methods look like.

```
public abstract class Array : ICloneable, IList, ICollection, IEnumerable,
   IStructuralComparable, IStructuralEquatable {

   public static void  Sort<T>(T[] array);
   public static void  Sort<T>(T[] array, IComparer<T> comparer);

   public static Int32 BinarySearch<T>(T[] array, T value);
   public static Int32 BinarySearch<T>(T[] array, T value,
      IComparer<T> comparer);
   ...
}
```

Here is code that demonstrates how to use some of these methods.

```
public static void Main() {
   // Create & initialize a byte array
   Byte[] byteArray = new Byte[] { 5, 1, 4, 2, 3 };

   // Call Byte[] sort algorithm
   Array.Sort<Byte>(byteArray);

   // Call Byte[] binary search algorithm
   Int32 i = Array.BinarySearch<Byte>(byteArray, 1);
   Console.WriteLine(i);   // Displays "0"
}
```

Generics Infrastructure

Generics were added to version 2.0 of the CLR, and it was a major task that required many people working for quite some time. Specifically, to make generics work, Microsoft had to do the following:

- Create new Intermediate Language (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 (C#, Microsoft Visual Basic .NET, etc.) 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. Also, new reflection emit members had to be defined so that developers could create generic type and method definitions at run time.
- Modify the debugger to show and manipulate generic types, members, fields, and local variables.
- Modify the Microsoft Visual Studio IntelliSense feature to show specific member prototypes when using a generic type or a method with a specific data type.

Now let's spend some time discussing how the CLR handles generics internally. This information could impact how you architect and design a generic algorithm. It could also impact your decision to use an existing generic algorithm or not.

Open and Closed Types

In various chapters throughout this book, I have discussed how the CLR creates an internal data structure for each and every type in use by an application. These data structures are called *type objects*. Well, 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 (classes), value types (structs), interface types, and delegate types. However, a type with generic type parameters is called an open type, and the CLR does not allow any instance of an *open type* to be constructed (similar to how the CLR prevents an instance of an interface type from being 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. However, it is possible for code referencing a generic type to leave some generic type arguments unspecified. This creates a new open type object in the CLR, and instances of this type cannot be created. The following code should make this clear.

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

// A partially specified open type
internal sealed class DictionaryStringKey<TValue> :
   Dictionary<String, TValue> {
}

public static class Program {
   public static void Main() {
      Object o = null;

      // Dictionary<,> is an open type having 2 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);

      // Prove it actually worked
      Console.WriteLine("Object type=" + 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 I compile the preceding code and run it, I get the following output.

```
Cannot create an instance of System.Collections.Generic.
Dictionary`2[TKey,TValue] because Type.ContainsGenericParameters is true.

Cannot create an instance of DictionaryStringKey`1[TValue] because
Type.ContainsGenericParameters is true.

Created instance of DictionaryStringKey`1[System.Guid]
Object type=DictionaryStringKey`1[System.Guid]
```

As you can see, `Activator`'s `CreateInstance` method throws an `ArgumentException` when you ask it to construct an instance of an open type. In fact, the exception's string message indicates that the type still contains some generic parameters.

In the output, you'll notice that the type names end with a backtick (`) followed by a number. The number indicates the type's *arity*, which indicates the number of type parameters required by the type. For example, the `Dictionary` class has an arity of 2 because it requires that types be specified for `TKey` and `TValue`. The `DictionaryStringKey` class has an arity of 1 because it requires just one type to be specified for `TValue`.

I should also point out that the CLR allocates a type's static fields inside the type object (as discussed in Chapter 4, "Type Fundamentals"). So each closed type has its own static fields. In other words, if `List<T>` defined any static fields, these fields are not shared between a `List<DateTime>` and a `List<String>`; each closed type object has its own static fields. Also, if a generic type defines a static constructor (discussed in Chapter 8, "Methods"), this constructor will execute once per closed type. Sometimes people define a static constructor on a generic type to ensure that the type arguments will meet certain criteria. For example, if you wanted to define a generic type that can be used only with enumerated types, you could do the following.

```
internal sealed class GenericTypeThatRequiresAnEnum<T> {
   static GenericTypeThatRequiresAnEnum() {
      if (!typeof(T).IsEnum) {
         throw new ArgumentException("T must be an enumerated type");
      }
   }
}
```

The CLR has a feature called constraints that offers a better way for you to define a generic type indicating what type arguments are valid for it. I'll discuss constraints later in this chapter. Unfortunately, constraints do not support the ability to limit a type argument to enumerated types only, which is why the previous example requires a static constructor to ensure that the type is an enumerated type.

Generic Types and Inheritance

A generic type is a type, and as such, it can be derived from any other type. When you use a 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 the generic type was derived from. In other words, because `List<T>` is derived from `Object`, `List<String>` and `List<Guid>` are also derived from `Object`. Similarly, because `DictionaryStringKey<TValue>` is derived from `Dictionary<String, TValue>`, `DictionaryStringKey<Guid>` is also derived from `Dictionary<String, Guid>`. Understanding that specifying type arguments doesn't have anything to do with inheritance hierarchies will help you to recognize what kind of casting you can and can't do.

For example, if a linked-list node class is defined like this:

```
internal sealed class Node<T> {
   public T m_data;
   public Node<T> m_next;

   public Node(T data) : this(data, null) {
   }

   public Node(T data, Node<T> next) {
      m_data = data; m_next = next;
   }

   public override String ToString() {
      return m_data.ToString() +
         ((m_next != null) ? m_next.ToString() : String.Empty);
   }
}
```

then I can write some code to build up a linked list that would look something like the following.

```
private static void SameDataLinkedList() {
   Node<Char> head = new Node<Char>('C');
   head = new Node<Char>('B', head);
   head = new Node<Char>('A', head);
   Console.WriteLine(head.ToString());  // Displays "ABC"
}
```

In the `Node` class just shown, the `m_next` field must refer to another node that has the same kind of data type in its `m_data` field. This means that the linked list must contain nodes in which all data items are of the same type (or derived type). For example, I can't use the `Node` class to create a linked list in which one element contains a `Char`, another element contains a `DateTime`, and another

element contains a `String`. Well, I could if I use `Node<Object>` everywhere, but then I would lose compile-time type safety, and value types would get boxed.

So a better way to go would be to define a non-generic `Node` base class and then define a generic `TypedNode` class (using the `Node` class as a base class). Now, I can have a linked list in which each node can be of a specific data type (not `Object`), get compile-time type safety, and avoid the boxing of value types. Here are the new class definitions.

```
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);
   }
}
```

I can now write code to create a linked list in which each node is a different data type. The code could look something like the following.

```
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());
}
```

Generic Type Identity

Sometimes generic syntax confuses developers. After all, there can be a lot of less-than (<) and greater-than (>) signs sprinkled throughout your source code, and this hurts readability. To improve syntax, some developers define a new non-generic class type that is derived from a generic type and that specifies all of the type arguments. For example, to simplify code like this:

```
List<DateTime> dtl = new List<DateTime>();
```

some developers might first define a class like the following.

```
internal sealed class DateTimeList : List<DateTime> {
   // No need to put any code in here!
}
```

Now, the code that creates a list can be rewritten more simply (without less-than and greater-than signs) like the following.

```
DateTimeList dtl = new DateTimeList();
```

Although this seems like a convenience, especially if you use the new type for parameters, local variables, and fields, you should never define a new class explicitly for the purpose of making your source code easier to read. The reason is because you lose type identity and equivalence, as you can see in the following code.

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

When the preceding code runs, `sameType` will be initialized to `false` because you are comparing two different type objects. This also means that a method prototyped as accepting a `DateTimeList` will not be able to have a `List<DateTime>` passed to it. However, a method prototyped as accepting a `List<DateTime>` can have a `DateTimeList` passed to it because `DateTimeList` is derived from `List<DateTime>`. Programmers may become easily confused by all of this.

Fortunately, 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.

```
using DateTimeList = System.Collections.Generic.List<System.DateTime>;
```

Here, the `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.DateTime>`. This 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, `sameType` will be initialized to `true`.

```
Boolean sameType = (typeof(List<DateTime>) == typeof(DateTimeList));
```

As another convenience, you can use C#'s implicitly typed local variable feature, where the compiler infers the type of a method's local variable from the type of the expression you are assigning to it.

```
using System;
using System.Collections.Generic;
...
internal sealed class SomeType {
   private static void SomeMethod () {
```

```
        // Compiler infers that dtl is of type
        // System.Collections.Generic.List<System.DateTime>
        var dtl = new List<DateTime>();
        ...
    }

}
```

Code Explosion

When a method that uses generic type parameters is JIT-compiled, the CLR takes the method's IL, substitutes the specified type arguments, and then creates native code that is specific to that method operating on the specified data types. This is exactly what you want and is one of the main features of generics. However, there is a downside to this: the CLR keeps generating native code for every method/type combination. This is referred to as *code explosion*. This can end up increasing the application's working set substantially, thereby hurting performance.

Fortunately, the 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 one assembly uses `List<DateTime>`, and a completely different assembly (loaded in the same AppDomain) also uses `List<DateTime>`, the CLR will compile the methods for `List<DateTime>` just once. This reduces code explosion substantially.

The CLR has another optimization: the CLR considers all reference type arguments to be identical, and so again, the code can be shared. For example, the code compiled by the CLR for `List<String>`'s methods can be used for `List<Stream>`'s methods, because `String` and `Stream` are both reference types. In fact, for any reference type, the same code will be used. The CLR can perform this optimization because all reference type arguments or variables are really just pointers (all 32 bits on a 32-bit Windows system and 64 bits on a 64-bit Windows system) to objects on the heap, and object pointers are all manipulated in the same way.

But if any type argument is a value type, the CLR must produce native code specifically for that value type. The reason is because value types can vary in size. And even if two value types are the same size (such as `Int32` and `UInt32`, which are both 32 bits), the CLR still can't share the code because different native CPU instructions can be used to manipulate these values.

Generic Interfaces

Obviously, the ability to define generic reference and value types was the main feature of generics. However, it was critical for the CLR to also allow generic interfaces. Without generic interfaces, any time you tried to manipulate a value type by using a non-generic interface (such as `IComparable`), boxing and a loss of compile-time type safety would happen again. This would severely limit the usefulness of generic types. And so the CLR does support generic interfaces. 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. Let's look at some examples.

Here is the definition of a generic interface that ships as part of the FCL (in the `System.Collections.Generic` namespace).

```
public interface IEnumerator<T> : IDisposable, IEnumerator {
   T Current { get; }
}
```

Here is an example of a type that implements this generic interface and that specifies type arguments. Notice that a `Triangle` object can enumerate a set of `Point` objects. Also note that the `Current` property is of the `Point` data type.

```
internal sealed class Triangle : IEnumerator<Point> {
   private Point[] m_vertices;

   // IEnumerator<Point>'s Current property is of type Point
   public Point Current { get { ... } }

   ...
}
```

Now let's look at an example of a type that implements the same generic interface but with the type arguments left unspecified.

```
internal sealed class ArrayEnumerator<T> : IEnumerator<T> {
   private T[] m_array;

   // IEnumerator<T>'s Current property is of type T
   public T Current { get { ... } }

   ...
}
```

Notice that an `ArrayEnumerator` object can enumerate a set of T objects (where T is unspecified allowing code using the generic `ArrayEnumerator` type to specify a type for T later). Also note that the `Current` property is now of the unspecified data type T. Much more information about generic interfaces is presented in Chapter 13, "Interfaces."

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. As discussed in Chapter 17, "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 compiler defines the delegate class's methods, and the type parameters are applied to any methods having parameters/return types of the specified type parameter.

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

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

the compiler turns that into a class that logically looks like the following.

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

Note It is recommended that you use the generic `Action` and `Func` delegates that come predefined in the Framework Class Library (FCL) wherever possible. I describe these delegate types in the "Enough with the Delegate Definitions Already (Generic Delegates)" section of Chapter 17.

Delegate and Interface Contra-variant and Covariant Generic Type Arguments

Each of a delegate's generic type parameters can be marked as covariant or contra-variant. This feature allows you to cast a variable of a generic delegate type to the *same delegate type* where the generic parameter types differ. A generic type parameter can be any one of the following:

- **Invariant** Meaning that the generic type parameter cannot be changed. I have shown only invariant generic type parameters so far in this chapter.
- **Contra-variant** Meaning that the generic type parameter can change from a class to a class derived from it. In C#, you indicate contra-variant generic type parameters with the `in` keyword. Contra-variant generic type parameters can appear only in input positions such as a method's argument.
- **Covariant** Meaning that the generic type argument can change from a class to one of its base classes. In C#, you indicate covariant generic type parameters with the `out` keyword. Covariant generic type parameters can appear only in output positions such as a method's return type.

For example, let's say that the following delegate type definition exists (which, by the way, it does).

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

Here, the generic type parameter `T` is marked with the `in` keyword, making it contra-variant; and the generic type parameter `TResult` is marked with the `out` keyword, making it covariant.

So now, if I have a variable declared as follows.

```
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("");
```

What this is saying is that fn1 refers to a function that accepts an Object and returns an ArgumentException. The fn2 variable wants to refer to a method that takes a String and returns an Exception. Because you can pass a String to a method that wants an Object (because String is derived from Object), and because you can take the result of a method that returns an ArgumentException and treat it as an Exception (because Exception is a base class of ArgumentException), the code above compiles and is known at compile time to preserve type safety.

Note Variance applies only if the compiler can verify that a reference conversion exists between types. In other words, variance is not possible for value types because boxing would be required. In my opinion, this restriction is what makes these variance features not that useful. For example, if I have the following method.

```
void ProcessCollection(IEnumerable<Object> collection) { ... }
```

I can't call it passing in a reference to a List<DateTime> object because a reference conversion doesn't exist between the DateTime value type and Object even though DateTime is derived from Object. You solve this problem by declaring ProcessCollection as follows.

```
void ProcessCollection<T>(IEnumerable<T> collection) { ... }
```

Plus, the big benefit of ProcessCollection(IEnumerable<Object> collection) is that there is only one version of the JITted code. However, with ProcessCollection<T>-(IEnumerable<T>collection), there is also only one version of the JITted code shared by all Ts that are reference types. You do get other versions of JITted code for Ts that are value types, but now you can at least call the method passing it a collection of value types.

Also, variance is not allowed on a generic type parameter if an argument of that type is passed to a method by using the out or ref keyword. For example, the line of code below causes the compiler to generate the following error message: Invalid variance: The 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 types, it is recommended to always specify the in and out keywords for contra-variance and covariance whenever possible, because doing this has no ill effects and enables your delegate to be used in more scenarios.

Like delegates, an interface with generic type parameters can have its type parameters be contra-variant or covariant. Here is an example of an interface with a covariant generic type parameter.

```
public interface IEnumerator<out T> : IEnumerator {
   Boolean MoveNext();
   T Current { get; }
}
```

Because T is covariant, it is possible to have the following code compile and run successfully.

```
// This method accepts an IEnumerable of any reference type
Int32 Count(IEnumerable<Object> collection) { ... }

...
// The call below passes an IEnumerable<String> to Count
Int32 c = Count(new[] { "Grant" });
```

Important Sometimes developers ask why they must explicitly put `in` or `out` on generic type parameters. They think the compiler should be able to examine the delegate or interface declaration and automatically detect what generic type parameters can be contra-variant and covariant. Although it is true that the compiler could detect this automatically, the C# team believes that you are declaring a contract and that you should be explicit about what you want to allow. For example, it would be bad if the compiler determined that a generic type parameter could be contra-variant and then, in the future, you added a member to an interface that had the type parameter used in an output position. The next time you compile, the compiler would determine that the type parameter should be invariant, but all code sites that reference the other members might now produce errors if they had used the fact that the type parameter had been contra-variant.

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 old contract.

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 type, 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 types, or local variables.

Here is a somewhat contrived example of a type that defines a type parameter and a method that has its very own type parameter.

```
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 `Swap` method.

```
private static void Swap<T>(ref T o1, ref T o2) {
   T temp = o1;
   o1 = o2;
   o2 = temp;
}
```

Code can now call `Swap` like as follows.

```
private static void CallingSwap() {
   Int32 n1 = 1, n2 = 2;
   Console.WriteLine("n1={0}, n2={1}", n1, n2);
   Swap<Int32>(ref n1, ref n2);
   Console.WriteLine("n1={0}, n2={1}", n1, n2);

   String s1 = "Aidan", s2 = "Grant";
   Console.WriteLine("s1={0}, s2={1}", s1, s2);
   Swap<String>(ref s1, ref s2);
   Console.WriteLine("s1={0}, s2={1}", s1, s2);
}
```

Using generic types with methods that take `out` and `ref` parameters can be particularly interesting because the variable you pass as an `out/ref` argument must be the same type as the method's parameter to avoid a potential type safety exploit. This issue related to `out/ref` parameters is discussed toward the end of the "Passing Parameters by Reference to a Method" section in Chapter 9, "Parameters." In fact, the `Interlocked` class's `Exchange` and `CompareExchange` methods offer generic overloads for precisely this reason.[1]

[1] The `where` clause will be explained in the "Verifiability and Constraints" section later in this chapter.

```
public static class Interlocked {
   public static T Exchange<T>(ref T location1, T value) where T: class;
   public static T CompareExchange<T>(
      ref T location1, T value, T comparand) where T: class;
}
```

Generic Methods and Type Inference

For many developers, the C# generic syntax can be confusing with all of its less-than and greater-than signs. 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 (or infer) the type to use automatically when calling a generic method. Here is some code that demonstrates type inference.

```
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, notice that the calls to Swap do not specify type arguments in less-than/greater-than signs. In the first call to Swap, the C# compiler was able to infer that `n1` and `n2` are `Int32`s, and therefore, it should call Swap by using an `Int32` type argument.

When performing type inference, C# uses the variable's data type, not the actual type of the object referred to by the variable. So in the second call to Swap, C# sees that `s1` is a `String` and `s2` is an `Object` (even though it happens to refer to a `String`). Because `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 the following message: `error CS0411: The type arguments for method 'Program.Swap<T>(ref T, ref T)' cannot be inferred from the usage. Try specifying the type arguments explicitly.`

A type can define multiple methods with one of its methods taking a specific data type and another taking a generic type parameter, as in the following example.

```
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>("Aidan");  // Calls Display<T>(T)
```

In the first call, the compiler could actually call either the `Display` method that takes a `String` or the generic `Display` method (replacing T with `String`). However, 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 (but with a `String` returned by `ToString`), there would have been infinite recursion.

The third call to `Display` specifies a generic type argument, `String`. This tells the compiler not to try to infer type arguments but instead to use the type arguments that I explicitly specified. In this case, the compiler also assumes that I must really want to call the generic `Display` method, so the generic `Display` will be called. Internally, the generic `Display` method will call `ToString` on the passed-in string, which results in a string that is then passed to the non-generic `Display` method.

Generics and Other Members

In C#, properties, indexers, events, operator methods, constructors, and finalizers cannot themselves have type parameters. However, they can be defined within a generic type, and the code in these members can use the type's type parameters.

C# doesn't allow these members to specify their own generic type parameters because Microsoft's C# team believes that developers would rarely have a need to use these members as generic. Furthermore, the cost of adding generic support to these members would be quite high in terms of designing adequate syntax into the language. For example, when you use a + operator in code, the compiler could call an operator overload method. There is no way to indicate any type arguments in your code along with the + operator.

Verifiability and Constraints

When compiling generic code, the C# compiler analyzes it and ensures that the code will work for any type that exists today or that may be defined in the future. Let's look at the following method.

```
private static Boolean MethodTakingAnyType<T>(T o) {
   T temp = o;
   Console.WriteLine(o.ToString());
   Boolean b = temp.Equals(o);
   return b;
}
```

This method declares a temporary variable (`temp`) of type T, and then the method performs a couple of variable assignments and a few method calls. This method works for any type. If T is a reference type, it works. If T is a value or enumeration type, it works. If T is an interface or delegate type, it works. This method works for all types that exist today or that will be defined tomorrow because

every type supports assignment and calls to methods defined by `Object` (such as `ToString` and `Equals`).

Now look at the following method.

```
private static T Min<T>(T o1, T o2) {
   if (o1.CompareTo(o2) < 0) return o1;
   return o2;
}
```

The `Min` method attempts to use the `o1` variable to call the `CompareTo` method. But there are lots of types that do not offer a `CompareTo` method, and therefore, the C# compiler can't compile this code and guarantee that this method would work for all types. If you attempt to compile the above code, the compiler issues the following message: `error CS1061: 'T' does not contain a definition for 'CompareTo' accepting a first argument of type 'T' could be found (are you missing a using directive or an assembly reference?)`

So it would seem that when using generics, you can declare variables of a generic type, perform some variable assignments, call methods defined by `Object`, and that's about it! This makes generics practically useless. Fortunately, compilers and the CLR support a mechanism called constraints that you can take advantage of to make generics useful again.

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 (in bold).

```
public static T Min<T>(T o1, T 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 because 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 example, the following code causes the compiler to issue the following message: `error CS0311: The type 'object' cannot be used as type parameter 'T' in the generic type or method 'SomeType.Min<T>(T, T)'. There is no implicit reference conversion from 'object' to 'System.IComparable<object>'.`

```
private static void CallMin() {
   Object o1 = "Jeff", o2 = "Richter";
   Object oMin = Min<Object>(o1, o2);  // Error CS0311
}
```

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.

Now that you have a sense of what constraints are and how they work, we'll start to look a little deeper into them. Constraints can be applied to a generic type's type parameters as well as to a generic method's type parameters (as shown in the `Min` method). The CLR doesn't allow overloading based on type parameter names or constraints; you can overload types or methods based only on arity. The following examples show what I mean.

```
// 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<T3, T4> {}

internal sealed class AnotherType {
   // It is OK to define the following methods:
   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: conflicts with M<T1, T2>
   private static void M<T3, T4>() {}
}
```

When overriding a virtual generic method, the overriding method must specify the same number of type parameters, and these type parameters will inherit the constraints specified on them by the base class's method. 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. Here is an example that demonstrates this rule by using virtual methods.

```
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
      { }
}
```

Attempting to compile the preceding code causes the compiler to issue the following message: `error CS0460: Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly.` If we remove the two `where` lines from the `Derived` class's `M<T3, T4>` method, the code will compile just fine. 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 (or even specify) constraints.

Now let's talk about the different kinds of constraints the compiler/CLR allows you to apply to a type parameter. A type parameter can be constrained by using a *primary constraint*, a *secondary constraint*, and/or a *constructor constraint*. I'll talk about these three kinds of constraints in the next three sections.

Primary Constraints

A type parameter can specify zero primary constraints or one primary constraint. A primary constraint can be a 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.Enum`, 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 constraint type. For example, see the following generic class.

```
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` (defined in the `System.IO` namespace). This tells the compiler that code using `PrimaryConstraintOfStream` must specify a type argument of `Stream` or a type derived from `Stream` (such as `FileStream`). If a type parameter doesn't specify a primary constraint, `System.Object` is assumed. However, the C# compiler issues an error message (`error CS0702: Constraint cannot be special class 'object'`) 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 a reference type. Any class type, interface type, delegate type, or array type satisfies this constraint. For example, see the following generic class.

```
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 preceding code 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>>`. Nullable types are discussed in Chapter 19, "Nullable Value Types."

Here is an example class that constrains its type parameter by using the `struct` constraint.

```
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.

Secondary Constraints

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 that implements the interface. And because you can specify multiple interface constraints, the type argument must specify a type that implements all of the interface constraints (and all of the primary constraints too, if specified). Chapter 13 discusses interface constraints in detail.

There is another kind of secondary constraint called a *type parameter constraint* (sometimes referred to as a *naked type constraint*). This kind of constraint is used much less often than an interface constraint. It allows a generic type or 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.

```
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 type 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 argument 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. Because 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. Because 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>. Because 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. Because String is not compatible with Exception, the fifth call doesn't adhere to the 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'.`

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. Here is an example class that constrains its type parameter by using the constructor constraint.

```
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 this 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.

Sometimes developers would like to declare a type parameter by using a constructor constraint whereby the constructor takes various parameters itself. As of now, the CLR (and therefore the C# compiler) supports only parameterless constructors. Microsoft feels that this will be good enough for almost all scenarios, and I agree.

Other Verifiability Issues

In the remainder of this section, I'd like to point out a few other code constructs that have unexpected behavior when used with generics due to verifiability issues and how constraints can be used to make the code verifiable again.

Casting a Generic Type Variable

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

```
private static void CastingAGenericTypeVariable1<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.

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

Although this code will now compile, it is still possible for the CLR to throw an `InvalidCastException` at run time.

If you are trying to cast to a reference type, you can also use the C# `as` operator. Here is code modified to use the `as` operator with `String` (because `Int32` is a value type).

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

Setting a Generic Type Variable to a Default Value

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

```
private static void SettingAGenericTypeVariableToNull<T>() {
   T temp = null;    // CS0403 - Cannot convert null to type parameter 'T' because it could
                     // be a non-nullable value type. Consider using 'default(T)' instead
}
```

Because 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.

Microsoft's 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.

```
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.

Comparing a Generic Type Variable with `null`

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

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

Because T is unconstrained, it could be a reference type or a value type. If T is a value type, `obj` can never be `null`. Normally, you'd expect the C# compiler to issue an error because of this. However, 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 (because it is always true), and it will emit the code inside the `if`'s braces.

By the way, if T had been constrained to a `struct`, the C# compiler would issue an error because you shouldn't be writing code that compares a value type variable with `null` because the result is always the same.

Comparing Two Generic Type Variables with Each Other

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 were constrained to `class`, this code would compile, and the == operator would return `true` if the variables referred to the same object, checking for exact identity. Note that if T were constrained to a reference type that overloaded the `operator ==` method, the compiler would emit calls to this method when it sees the == operator. Obviously, this whole discussion applies to uses of the != operator too.

When you write code to compare the primitive value types—`Byte`, `Int32`, `Single`, `Decimal`, etc.—the C# compiler knows how to emit the right code. However, for non-primitive value types, the C# compiler doesn't know how to emit the code to do comparisons. So if `ComparingTwoGenericTypeVariables` method's T were constrained to `struct`, the compiler would issue an error. And you're not allowed to constrain a type parameter to a specific value type because it is implicitly sealed, and therefore no types exist that are derived from the value type. Allowing this would make the generic method constrained to a specific type, and the C# compiler doesn't allow this because it is more efficient to just make a non-generic method.

Using Generic Type Variables as Operands

Finally, it should be noted that there are a lot of issues about using operators with generic type operands. In Chapter 5, I talked about C# and how it handles its primitive types: `Byte`, `Int16`, `Int32`, `Int64`, `Decimal`, and so on. In particular, I mentioned that C# knows how to interpret operators (such as +, -, *, and /) when applied to the primitive types. Well, these operators 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. Here is an example of a generic method that I'd like to write.

```
private static T Sum<T>(T num) where T : struct {
   T sum = default(T) ;
   for (T n = default(T) ; n < num ; n++)
      sum += n;
   return sum;
}
```

I've done everything possible to try to get this method to compile. I've constrained T to `struct`, and I'm using `default(T)` to initialize `sum` and `n` to 0. But when I compile this code, I get the following three errors.

- `error CS0019: Operator '<' cannot be applied to operands of type 'T' and 'T'`
- `error CS0023: Operator '++' cannot be applied to operand of type 'T'`
- `error CS0019: Operator '+=' cannot be applied to operands of type 'T' and 'T'`

This is a severe limitation on the CLR's generic support, and many developers (especially in the scientific, financial, and mathematical world) are very disappointed by this limitation. Many people have tried to come up with techniques to work around this limitation by using reflection (see Chapter 23, "Assembly Loading and Reflection"), the `dynamic` primitive type (see Chapter 5), operator overloading, and so on. But all of these cause a severe performance penalty or hurt readability of the code substantially. Hopefully, this is an area that Microsoft will address in a future version of the CLR and the compilers.