C

Version 2.0 Specification

July 2003

Table of Contents

Chapter

19. Introduction to C# 2.0

C# 2.0 introduces several language extensions, the most important of which are Generics, Anonymous Methods, Iterators, and Partial Types.

- Generics permit classes, structs, interfaces, delegates, and methods to be parameterized by the types of data they store and manipulate. Generics are useful because they provide stronger compile-time type checking, require fewer explicit conversions between data types, and reduce the need for boxing operations and run-time type checks.
- Anonymous methods allow code blocks to be written "in-line" where delegate values are expected. Anonymous methods are similar to lambda functions in the Lisp programming language. C# 2.0 supports the creation of "closures" where anonymous methods access surrounding local variables and parameters.
- Iterators are methods that incrementally compute and yield a sequence of values. Iterators make it easy for a type to specify how the `foreach` statement will iterate over its elements.
- Partial types allow classes, structs, and interfaces to be broken into multiple pieces stored in different source files for easier development and maintenance. Additionally, partial types allow separation of machine-generated and user-written parts of types so that it is easier to augment code generated by a tool.

This chapter gives an introduction to these new features. Following the introduction are four chapters that provide a complete technical specification of the features.

The language extensions in C# 2.0 were designed to ensure maximum compatibility with existing code. For example, even though C# 2.0 gives special meaning to the words `where`, `yield`, and `partial` in certain contexts, these words can still be used as identifiers. Indeed, C# 2.0 adds no new keywords as such keywords could conflict with identifiers in existing code.

19.1 Generics

Generics permit classes, structs, interfaces, delegates, and methods to be parameterized by the types of data they store and manipulate. C# generics will be immediately familiar to users of generics in Eiffel or Ada, or to users of C++ templates, though they do not suffer many of the complications of the latter.

19.1.1 Why generics?

Without generics, general purpose data structures can use type `object` to store data of any type. For example, the following simple `Stack` class stores its data in an `object` array, and its two methods, `Push` and `Pop`, use `object` to accept and return data, respectively:

```
public class Stack
{
   object[] items;
   int count;

   public void Push(object item) {...}

   public object Pop() {...}
}
```

While the use of type `object` makes the `Stack` class very flexible, it is not without drawbacks. For example, it is possible to push a value of any type, such a `Customer` instance, onto a stack. However, when a value is

retrieved, the result of the Pop method must explicitly be cast back to the appropriate type, which is tedious to write and carries a performance penalty for run-time type checking:

```
Stack stack = new Stack();
stack.Push(new Customer());
Customer c = (Customer)stack.Pop();
```

If a value of a value type, such as an int, is passed to the Push method, it is automatically boxed. When the int is later retrieved, it must be unboxed with an explicit type cast:

```
Stack stack = new Stack();
stack.Push(3);
int i = (int)stack.Pop();
```

Such boxing and unboxing operations add performance overhead since they involve dynamic memory allocations and run-time type checks.

A further issue with the Stack class is that it is not possible to enforce the kind of data placed on a stack. Indeed, a Customer instance can be pushed on a stack and then accidentally cast it to the wrong type after it is retrieved:

```
Stack stack = new Stack();
stack.Push(new Customer());
string s = (string)stack.Pop();
```

While the code above is an improper use of the Stack class, the code is technically speaking correct and a compile-time error is not reported. The problem does not become apparent until the code is executed, at which point an InvalidCastException is thrown.

The Stack class would clearly benefit from the ability to specify its element type. With generics, that becomes possible.

19.1.2 Creating and using generics

Generics provide a facility for creating types that have ***type parameters***. The example below declares a generic Stack class with a type parameter T. The type parameter is specified in < and > delimiters after the class name. Rather than forcing conversions to and from object, instances of Stack<T> accept the type for which they are created and store data of that type without conversion. The type parameter T acts as a placeholder until an actual type is specified at use. Note that T is used as the element type for the internal items array, the type for the parameter to the Push method, and the return type for the Pop method:

```
public class Stack<T>
{
   T[] items;
   int count;

   public void Push(T item) {...}

   public T Pop() {...}
}
```

When the generic class Stack<T> is used, the actual type to substitute for T is specified. In the following example, int is given as the ***type argument*** for T:

```
Stack<int> stack = new Stack<int>();
stack.Push(3);
int x = stack.Pop();
```

The Stack<int> type is called a ***constructed type***. In the Stack<int> type, every occurrence of T is replaced with the type argument int. When an instance of Stack<int> is created, the native storage of the items array is an int[] rather than object[], providing substantial storage efficiency compared to the non-generic Stack. Likewise, the Push and Pop methods of a Stack<int> operate on int values, making it a compile-time error

to push values of other types onto the stack, and eliminating the need to explicitly cast values back to their original type when they're retrieved.

Generics provide strong typing, meaning for example that it is an error to push an `int` onto a stack of `Customer` objects. Just as a `Stack<int>` is restricted to operate only on `int` values, so is `Stack<Customer>` restricted to `Customer` objects, and the compiler will report errors on the last two lines of the following example:

```
Stack<Customer> stack = new Stack<Customer>();
stack.Push(new Customer());
Customer c = stack.Pop();
stack.Push(3);                // Type mismatch error
int x = stack.Pop();          // Type mismatch error
```

Generic type declarations may have any number of type parameters. The `Stack<T>` example above has only one type parameter, but a generic `Dictionary` class might have two type parameters, one for the type of the keys and one for the type of the values:

```
public class Dictionary<K,V>
{
   public void Add(K key, V value) {...}

   public V this[K key] {...}
}
```

When `Dictionary<K,V>` is used, two type arguments would have to be supplied:

```
Dictionary<string,Customer> dict = new Dictionary<string,Customer>();
dict.Add("Peter", new Customer());
Customer c = dict["Peter"];
```

19.1.3 Generic type instantiations

Similar to a non-generic type, the compiled representation of a generic type is intermediate language (IL) instructions and metadata. The representation of the generic type of course also encodes the existence and use of type parameters.

The first time an application creates an instance of a constructed generic type, such as `Stack<int>`, the just-in-time (JIT) compiler of the .NET Common Language Runtime converts the generic IL and metadata to native code, substituting actual types for type parameters in the process. Subsequent references to that constructed generic type then use the same native code. The process of creating a specific constructed type from a generic type is known as a ***generic type instantiation***.

The .NET Common Language Runtime creates a specialized copy of the native code for each generic type instantiation with a value type, but shares a single copy of the native code for all reference types (since, at the native code level, references are just pointers with the same representation).

19.1.4 Constraints

Commonly, a generic class will do more than just store data based on a type parameter. Often, the generic class will want to invoke methods on objects whose type is given by a type parameter. For example, an `Add` method in a `Dictionary<K,V>` class might need to compare keys using a `CompareTo` method:

```
public class Dictionary<K,V>
{
   public void Add(K key, V value)
   {
      ...
```

```
            if (key.CompareTo(x) < 0) {...}     // Error, no CompareTo method
            ...
        }
    }
```

Since the type argument specified for `K` could be any type, the only members that can be assumed to exist on the `key` parameter are those declared by type `object`, such as `Equals`, `GetHashCode`, and `ToString`; a compile-time error therefore occurs in the example above. It is of course possible to cast the `key` parameter to a type that contains a `CompareTo` method. For example, the `key` parameter could be cast to `IComparable`:

```
public class Dictionary<K,V>
{
    public void Add(K key, V value)
    {
        ...
        if (((IComparable)key).CompareTo(x) < 0) {...}
        ...
    }
}
```

While this solution works, it requires a dynamic type check at run-time, which adds overhead. It furthermore defers error reporting to run-time, throwing an `InvalidCastException` if a key doesn't implement `IComparable`.

To provide stronger compile-time type checking and reduce type casts, C# permits an optional list of ***constraints*** to be supplied for each type parameter. A type parameter constraint specifies a requirement that a type must fulfill in order to be used as an argument for that type parameter. Constraints are declared using the word `where`, followed by the name of a type parameter, followed by a list of class or interface types, or the constructor constraint `new()`.

In order for the `Dictionary<K,V>` class to ensure that keys always implement `IComparable`, the class declaration can specify a constraint for the type parameter `K`:

```
public class Dictionary<K,V> where K: IComparable
{
    public void Add(K key, V value)
    {
        ...
        if (key.CompareTo(x) < 0) {...}
        ...
    }
}
```

Given this declaration the compiler will ensure that any type argument supplied for `K` is a type that implements `IComparable`. Furthermore, it is no longer necessary to explicitly cast the key parameter to `IComparable` before calling the `CompareTo` method; all members of a type given as a constraint for a type parameter are directly available on values of that type parameter type.

For a given type parameter, it is possible to specify any number of interfaces as constraints, but no more than one class. Each constrained type parameter has a separate `where` clause. In the example below, the type parameter `K` has two interface constraints, while the type parameter `E` has a class constraint and a constructor constraint:

```
public class EntityTable<K,E>
    where K: IComparable<K>, IPersistable
    where E: Entity, new()
{
    public void Add(K key, E entity)
    {
        ...
```



```
            if (key.CompareTo(x) < 0) {...}
            ...
        }
    }
```

The constructor constraint, `new()`, in the example above ensures that a type used as a type argument for `E` has a public, parameterless constructor, and it permits the generic class to use `new E()` to create instances of that type.

Type parameter constrains should be used with care. While they provide stronger compile-time type checking and in some cases improve performance, they also restrict the possible uses of a generic type. For example, a generic class `List<T>` might constrain `T` to implement `IComparable` such that the list's `Sort` method can compare items. However, doing so would preclude use of `List<T>` for types that don't implement `IComparable`, even if the `Sort` method is never actually called in those cases.

19.1.5 Generic methods

In some cases a type parameter is not needed for an entire class, but only inside a particular method. Often, this occurs when creating a method that takes a generic type as a parameter. For example, when using the `Stack<T>` class described earlier, a common pattern might be to push multiple values in a row, and it might be convenient to write a method that does so in a single call. For a particular constructed type, such as `Stack<int>`, the method would look like this:

```
void PushMultiple(Stack<int> stack, params int[] values) {
   foreach (int value in values) stack.Push(value);
}
```

This method can be used to push multiple `int` values onto a `Stack<int>`:

```
Stack<int> stack = new Stack<int>();
PushMultiple(stack, 1, 2, 3, 4);
```

However, the method above only works with the particular constructed type `Stack<int>`. To have it work with any `Stack<T>`, the method must be written as a ***generic method***. A generic method has one or more type parameters specified in < and > delimiters after the method name. The type parameters can be used within the parameter list, return type, and body of the method. A generic `PushMultiple` method would look like this:

```
void PushMultiple<T>(Stack<T> stack, params T[] values) {
   foreach (T value in values) stack.Push(value);
}
```

Using this generic method, it is possible to push multiple items onto any `Stack<T>`. When calling a generic method, type arguments are given in angle brackets in the method invocation. For example:

```
Stack<int> stack = new Stack<int>();
PushMultiple<int>(stack, 1, 2, 3, 4);
```

This generic `PushMultiple` method is more reusable than the previous version, since it works on any `Stack<T>`, but it appears to be less convenient to call, since the desired `T` must be supplied as a type argument to the method. In many cases, however, the compiler can deduce the correct type argument from the other arguments passed to the method, using a process called ***type inferencing***. In the example above, since the first regular argument is of type `Stack<int>`, and the subsequent arguments are of type `int`, the compiler can reason that the type parameter must be `int`. Thus, the generic `PushMultiple` method can be called without specifying the type parameter:

```
Stack<int> stack = new Stack<int>();
PushMultiple(stack, 1, 2, 3, 4);
```

19.2 Anonymous methods

Event handlers and other callbacks are often invoked exclusively through delegates and never directly. Even so, it has thus far been necessary to place the code of event handlers and callbacks in distinct methods to which

delegates are explictly created. In contrast, ***anonymous methods*** allow the code associated with a delegate to be written "in-line" where the delegate is used, conveniently tying the code directly to the delegate instance. Besides this convenience, anonymous methods have shared access to the local state of the containing function member. To achieve the same state sharing using named methods requires "lifting" local variables into fields in instances of manually authored helper classes.

The following example shows a simple input form that contains a list box, a text box, and a button. When the button is clicked, an item containing the text in the text box is added to the list box.

```
class InputForm: Form
{
   ListBox listBox;
   TextBox textBox;
   Button addButton;

   public MyForm() {
      listBox = new ListBox(...);
      textBox = new TextBox(...);
      addButton = new Button(...);

      addButton.Click += new EventHandler(AddClick);
   }

   void AddClick(object sender, EventArgs e) {
      listBox.Items.Add(textBox.Text);
   }
}
```

Even though only a single statement is executed in response to the button's `Click` event, that statement must be extracted into a separate method with a full parameter list, and an `EventHandler` delegate referencing that method must be manually created. Using an anonymous method, the event handling code becomes significantly more succinct:

```
class InputForm: Form
{
   ListBox listBox;
   TextBox textBox;
   Button addButton;

   public MyForm() {
      listBox = new ListBox(...);
      textBox = new TextBox(...);
      addButton = new Button(...);

      addButton.Click += delegate {
         listBox.Items.Add(textBox.Text);
      };
   }
}
```

An anonymous method consists of the keyword `delegate`, an optional parameter list, and a statement list enclosed in `{` and `}` delimiters. The anonymous method in the previous example doesn't use the parameters supplied by the delegate, and it can therefore omit the parameter list. To gain access to the parameters, the anonymous method can include a parameter list:

```
addButton.Click += delegate(object sender, EventArgs e) {
   MessageBox.Show(((Button)sender).Text);
};
```

In the previous examples, an implicit conversion occurs from the anonymous method to the `EventHandler` delegate type (the type of the `Click` event). This implict conversion is possible because the parameter list and return type of the delegate type are compatible with the anonymous method. The exact rules for compatibility are as follows:

- The parameter list of a delegate is compatible with an anonymous method if one of the following is true:
  - The anonymous method has no parameter list and the delegate has no `out` parameters.
  - The anonymous method includes a parameter list that exactly matches the delegate's parameters in number, types, and modifiers.
- The return type of a delegate is compatible with an anonymous method if one of the following is true:
  - The delegate's return type is `void` and the anonymous method has no `return` statements or only `return` statements with no expression.
  - The delegate's return type is not `void` and the expressions associated with all `return` statements in the anonymous method can be implicitly converted to the return type of the delegate.

Both the parameter list and the return type of a delegate must be compatible with an anonymous method before an implicit conversion to that delegate type can occur.

The following example uses anonymous methods to write functions "in-line." The anonymous methods are passed as parameters of a `Function` delegate type.

```
using System;

delegate double Function(double x);

class Test
{
   static double[] Apply(double[] a, Function f) {
      double[] result = new double[a.Length];
      for (int i = 0; i < a.Length; i++) result[i] = f(a[i]);
      return result;
   }

   static double[] MultiplyAllBy(double[] a, double factor) {
      return Apply(a, delegate(double x) { return x * factor; });
   }

   static void Main() {
      double[] a = {0.0, 0.5, 1.0};

      double[] squares = Apply(a, delegate(double x) { return x * x; });

      double[] doubles = MultiplyAllBy(a, 2.0);
   }
}
```

The `Apply` method applies a given `Function` to the elements of a `double[]`, returning a `double[]` with the results. In the `Main` method, the second parameter passed to `Apply` is an anonymous method that is compatible with the `Function` delegate type. The anonymous method simply returns the square of its argument, and thus the result of that `Apply` invocation is a `double[]` containing the squares of the values in `a`.

The `MultiplyAllBy` method returns a `double[]` created by multiplying each of the values in the argument array `a` by a given `factor`. In order to produce its result, `MultiplyAllBy` invokes the `Apply` method, passing an anonymous method that multiplies the argument `x` by `factor`.

Local variables and parameters whose scope contains an anonymous method are called ***outer variables*** of the anonymous method. In the `MultiplyAllBy` method, `a` and `factor` are outer variables of the anonymous method passed to `Apply`, and because the anonymous method references `factor`, `factor` is said to have been ***captured*** by the anonymous method. Ordinarily, the lifetime of a local variable is limited to execution of the block or statement with which it is associated. However, the lifetime of a captured outer variable is extended at least until the delegate referring to the anonymous method becomes eligible for garbage collection.

19.2.1 Method group conversions

As described in the previous section, an anonymous method can be implicitly converted to a compatible delegate type. C# 2.0 permits this same type of conversion for a method group, allowing explicit delegate instantiations to be omitted in almost all cases. For example, the statements

```
addButton.Click += new EventHandler(AddClick);
Apply(a, new Function(Math.Sin));
```

can instead be written

```
addButton.Click += AddClick;
Apply(a, Math.Sin);
```

When the shorter form is used, the compiler automatically infers which delegate type to instantiate, but the effects are otherwise the same as the longer form.

19.3 Iterators

The C# `foreach` statement is used to iterate over the elements of an ***enumerable*** collection. In order to be enumerable, a collection must have a parameterless `GetEnumerator` method that returns an ***enumerator***. Generally, enumerators are difficult to implement, but the task is significantly simplified with iterators.

An ***iterator*** is a statement block that ***yields*** an ordered sequence of values. An iterator is distinguished from a normal statement block by the presence of one or more `yield` statements:

- The `yield return` statement produces the next value of the iteration.
- The `yield break` statement indicates that the iteration is complete.

An iterator may be used as the body of a function member as long as the return type of the function member is one of the ***enumerator interfaces*** or one of the ***enumerable interfaces***:

- The enumerator interfaces are `System.Collections.IEnumerator` and types constructed from `System.Collections.Generic.IEnumerator<T>`.
- The enumerable interfaces are `System.Collections.IEnumerable` and types constructed from `System.Collections.Generic.IEnumerable<T>`.

It is important to understand that an iterator is not a kind of member, but is a means of implementing a function member. A member implemented via an iterator may be overridden or overloaded by other members which may or may not be implemented with iterators.

The following `Stack<T>` class implements its `GetEnumerator` method using an iterator. The iterator enumerates the elements of the stack in top to bottom order.

```
using System.Collections.Generic;

public class Stack<T>: IEnumerable<T>
{
   T[] items;
   int count;

   public void Push(T data) {...}

   public T Pop() {...}

   public IEnumerator<T> GetEnumerator() {
      for (int i = count – 1; i >= 0; --i) {
         yield return items[i];
      }
   }
}
```

The presence of the `GetEnumerator` method makes `Stack<T>` an enumerable type, allowing instances of `Stack<T>` to be used in a `foreach` statement. The following example pushes the values 0 through 9 onto an integer stack and then uses a `foreach` loop to display the values in top to bottom order.

```
using System;

class Test
{
   static void Main() {
      Stack<int> stack = new Stack<int>();
      for (int i = 0; i < 10; i++) stack.Push(i);
      foreach (int i in stack) Console.Write("{0} ", i);
      Console.WriteLine();
   }
}
```

The output of the example is:

```
9 8 7 6 5 4 3 2 1 0
```

The `foreach` statement implicitly calls a collection's parameterless `GetEnumerator` method to obtain an enumerator. There can only be one such parameterless `GetEnumerator` method defined by a collection, yet it is often appropriate to have multiple ways of enumerating, and ways of controlling the enumeration through parameters. In such cases, a collection can use iterators to implement properties or methods that return one of the enumerable interfaces. For example, `Stack<T>` might introduce two new properties, `TopToBottom` and `BottomToTop`, of type `IEnumerable<T>`:

```
using System.Collections.Generic;

public class Stack<T>: IEnumerable<T>
{
   T[] items;
   int count;

   public void Push(T data) {...}

   public T Pop() {...}

   public IEnumerator<T> GetEnumerator() {
      for (int i = count – 1; i >= 0; --i) {
         yield return items[i];
      }
   }

   public IEnumerable<T> TopToBottom {
      get {
         return this;
      }
   }

   public IEnumerable<T> BottomToTop {
      get {
         for (int i = 0; i < count; i++) {
            yield return items[i];
         }
      }
   }
}
```

The `get` accessor for the `TopToBottom` property just returns `this` since the stack itself is an enumerable. The `BottomToTop` property returns an enumerable implemented with a C# iterator. The following example shows how the properties can be used to enumerate stack elements in either order:

```
using System;
```

```
class Test
{
   static void Main() {
      Stack<int> stack = new Stack<int>();
      for (int i = 0; i < 10; i++) stack.Push(i);

      foreach (int i in stack.TopToBottom) Console.Write("{0} ", i);
      Console.WriteLine();

      foreach (int i in stack.BottomToTop) Console.Write("{0} ", i);
      Console.WriteLine();
   }
}
```

Of course, these properties can be used outside of a `foreach` statement as well. The following example passes the results of invoking the properties to a separate `Print` method. The example also shows an iterator used as the body of a `FromToBy` method that takes parameters:

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

class Test
{
   static void Print(IEnumerable<int> collection) {
      foreach (int i in collection) Console.Write("{0} ", i);
      Console.WriteLine();
   }

   static IEnumerable<int> FromToBy(int from, int to, int by) {
      for (int i = from; i <= to; i += by) {
         yield return i;
      }
   }

   static void Main() {
      Stack<int> stack = new Stack<int>();
      for (int i = 0; i < 10; i++) stack.Push(i);
      Print(stack.TopToBottom);
      Print(stack.BottomToTop);
      Print(FromToBy(10, 20, 2));
   }
}
```

The output of the example is:

```
9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9
10 12 14 16 18 20
```

The generic and non-generic enumerable interfaces contain a single member, a `GetEnumerator` method that takes no arguments and returns an enumerator interface. An enumerable acts as an *enumerator factory*. Properly implemented enumerables generate independent enumerators each time their `GetEnumerator` method is called. Assuming the internal state of the enumerable has not changed between two calls to `GetEnumerator`, the two enumerators returned should produce the same set of values in the same order. This should hold even if the lifetime of the enumerators overlap as in the following code sample:

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

class Test
{
   static IEnumerable<int> FromTo(int from, int to) {
      while (from <= to) yield return from++;
   }
```



```
   static void Main() {
      IEnumerable<int> e = FromTo(1, 10);
      foreach (int x in e) {
         foreach (int y in e) {
            Console.Write("{0,3} ", x * y);
         }
         Console.WriteLine();
      }
   }
}
```

The code above prints a simple multiplication table of the integers 1 through 10. Note that the `FromTo` method is invoked only once to generate the enumerable `e`. However, `e.GetEnumerator()` is invoked multiple times (by the `foreach` statements) to generate multiple equivalent enumerators. These enumerators all encapsulate the iterator code specified in the declaration of `FromTo`. Note that the iterator code modifies the `from` parameter. Nevertheless, the enumerators act independently because each enumerator is given *its own copy* of the `from` and `to` parameters. The sharing of transient state between enumerators is one of several common subtle flaws that should be avoided when implementing enumerables and enumerators. C# iterators are designed to help avoid these problems and to implement robust enumerables and enumerators in a simple intuitive way.

19.4 Partial types

While it is good programming practice to maintain all source code for a type in a single file, sometimes a type becomes large enough that this is an impractical constraint. Furthermore, programmers often use source code generators to produce the initial structure of an application, and then modify the resulting code. Unfortunately, when source code is emitted again sometime in the future, existing modifications are overwritten.

Partial types allow classes, structs, and interfaces to be broken into multiple pieces stored in different source files for easier development and maintenance. Additionally, partial types allow separation of machine-generated and user-written parts of types so that it is easier to augment code generated by a tool.

A new type modifier, `partial`, is used when defining a type in multiple parts. The following is an example of a partial class that is implemented in two parts. The two parts may be in different source files, for example because the first part is machine generated by a database mapping tool and the second part is manually authored:

```
public partial class Customer
{
   private int id;
   private string name;
   private string address;
   private List<Order> orders;

   public Customer() {
      ...
   }
}

public partial class Customer
{
   public void SubmitOrder(Order order) {
      orders.Add(order);
   }

   public bool HasOutstandingOrders() {
      return orders.Count > 0;
   }
}
```

When the two parts above are compiled together, the resulting code is the same as if the class had been written as a single unit:

```
public class Customer
{
   private int id;
   private string name;
   private string address;
   private List<Order> orders;

   public Customer() {
      ...
   }

   public void SubmitOrder(Order order) {
      orders.Add(order);
   }

   public bool HasOutstandingOrders() {
      return orders.Count > 0;
   }
}
```

All parts of a partial type must be compiled together such that the parts can be merged at compile-time. Partial types specifically do not allow already compiled types to be extended.

20. Generics

20.1 Generic class declarations

A generic class declaration is a declaration of a class that requires type parameters to be supplied in order to form actual types.

A class declaration may optionally define type parameters:

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `class` *identifier* *type-parameter-list*$_{opt}$ *class-base*$_{opt}$
type-parameter-constraints-clauses$_{opt}$ *class-body* `;`$_{opt}$

A class declaration may not supply *type-parameter-constraints-clauses* (§20.7) unless it also supplies a *type-parameter-list*.

A class declaration that supplies a *type-parameter-list* is a generic class declaration. Additionally, any class nested inside a generic class declaration or a generic struct declaration is itself a generic class declaration, since type parameters for the containing type must be supplied to create a constructed type.

Generic class declarations follow the same rules as normal class declarations except where noted, and particularly with regard to naming, nesting and the permitted access controls. Generic class declarations may be nested inside non-generic class declarations.

A generic class is referenced using a ***constructed type*** (§20.4). Given the generic class declaration

```
class List<T> {}
```

some examples of constructed types are `List<T>`, `List<int>` and `List<List<string>>`. A constructed type that uses one or more type parameters, such as `List<T>`, is called a ***open constructed type***. A constructed type that uses no type parameters, such as `List<int>`, is called a ***closed constructed type***.

Generic types may not be “overloaded”, that is the identifier of a generic type must be uniquely named within a scope in the same way as ordinary types.

```
class C {}
class C<V> {}        // Error, C defined twice
class C<U,V> {}      // Error, C defined twice
```

However, the type lookup rules used during unqualified type name lookup (§20.9.3) and member access (§20.9.4) do take the number of generic parameters into account.

20.1.1 Type parameters

Type parameters may be supplied on a class declaration. Each type parameter is a simple identifier which denotes a placeholder for a type argument that is supplied to create a constructed type. A type parameter is a formal placeholder for a type that will be supplied later. By constrast, a type argument (§20.5.1) is the actual type that is substituted for the type parameter when a constructed type is referenced.

type-parameter-list:
`<` *type-parameters* `>`

type-parameters:
type-parameter
type-parameters , *type-parameter*

type-parameter:
attributes$_{opt}$ *identifier*

Each type parameter in a class declaration defines a name in the declaration space (§3.3) of that class. Thus, it cannot have the same name as another type parameter or a member declared in that class. A type parameter cannot have the same name as the type itself.

The scope (§3.7) of a type parameter on a class includes the *class-base*, *type-parameter-constraints-clauses*, and *class-body*. Unlike members of a class, it does not extend to derived classes. Within its scope, a type parameter can be used as a type.

type:
value-type
reference-type
type-parameter

Since a type parameter can be instantiated with many different actual type arguments, type parameters have slightly different operations and restrictions than other types. These include:

- A type parameter cannot be used directly to declare a base class or interface (§20.1.3).
- The rules for member lookup on type parameters depend on the constraints, if any, applied to the type. They are detailed in §20.7.2.
- The available conversions for a type parameter depend on the constraints, if any, applied to the type. They are detailed in §20.7.4.
- The literal `null` cannot be converted to a type given by a type parameter, except if the type parameter is constrained by a class constraint (§20.7.4). However, a default value expression (§20.8.1) can be used instead. In addition, a value with a type given by a type parameter *can* be compared with `null` using `==` and `!=` (§20.8.4).
- A `new` expression (§20.8.2) can only be used with a type parameter if the type parameter is constrained by a *constructor-constraint* (§20.7).
- A type parameter cannot be used anywhere within an attribute.
- A type parameter cannot be used in a member access or type name to identify a static member or a nested type (§20.9.1, §20.9.4).
- In unsafe code, a type parameter cannot be used as an *unmanaged-type* (§18.2).

As a type, type parameters are purely a compile-time construct. At run-time, each type parameter is bound to a run-time type that was specified by supplying a type argument to the generic type declaration. Thus, the type of a variable declared with a type parameter will, at run-time, be a closed type (§20.5.2). The run-time execution of all statements and expressions involving type parameters uses the actual type that was supplied as the type argument for that parameter.

20.1.2 The instance type

Each class declaration has an associated constructed type, the ***instance type***. For a generic class declaration, the instance type is formed by creating a constructed type (§20.4) from the type declaration, with each of the supplied type arguments being the corresponding type parameter. Since the instance type uses the type parameters, it is only valid where the type parameters are in scope: inside the class declaration. The instance

type is the type of `this` for code written inside the class declaration. For non-generic classes, the instance type is simply the declared class. The following shows several class declarations along with their instance types:

```
class A<T>                    // instance type: A<T>
{
   class B {}                 // instance type: A<T>.B
   class C<U> {}              // instance type: A<T>.C<U>
}
class D {}                    // instance type: D
```

20.1.3 Base specification

The base class specified in a class declaration may be a constructed class type (§20.4). A base class may not be a type parameter on its own, though it may involve the type parameters that are in scope.

```
class Extend<V>: V {}         // Error, type parameter used as base class
```

A generic class declaration may not use `System.Attribute` as a direct or indirect base class.

The base interfaces specified in a class declaration may be constructed interface types (§20.4). A base interface may not be a type parameter on its own, though it may involve the type parameters that are in scope. The following code illustrates how a class can implement and extend constructed types:

```
class C<U,V> {}
interface I1<V> {}
class D: C<string,int>, I1<string> {}
class E<T>: C<int,T>, I1<T> {}
```

The base interfaces of a generic class declaration must satisfy the uniqueness rule described in §20.3.1.

Methods in a class that override or implement methods from a base class or interface must provide appropriate methods of specialized types. The following code illustrates how methods are overridden and implemented. This is explained further in §20.1.10.

```
class C<U,V>
{
   public virtual void M1(U x, List<V> y) {...}
}
interface I1<V>
{
   V M2(V);
}
class D: C<string,int>, I1<string>
{
   public override void M1(string x, List<int> y) {...}
   public string M2(string x) {...}
}
```

20.1.4 Members of generic classes

All members of a generic class may use type parameters from any enclosing class, either directly or as part of a constructed type. When a particular closed constructed type (§20.5.2) is used at run-time, each use of a type parameter is replaced with the actual type argument supplied to the constructed type. For example:

```
class C<V>
{
   public V f1;
   public C<V> f2 = null;
```



```
   public C(V x) {
      this.f1 = x;
      this.f2 = this;
   }
}

class Application
{
   static void Main() {
      C<int> x1 = new C<int>(1);
      Console.WriteLine(x1.f1);     // Prints 1

      C<double> x2 = new C<double>(3.1415);
      Console.WriteLine(x2.f1);     // Prints 3.1415
   }
}
```

Within instance function members, the type of `this` is the instance type (§20.1.2) of the declaration.

Apart from the use of type parameters as types, members in generic class declarations follow the same rules as members of non-generic classes. Additional rules that apply to particular kinds of members are discussed in the following sections.

20.1.5 Static fields in generic classes

A static variable in a generic class declaration is shared amongst all instances of the same closed constructed type (§20.5.2), but is not shared amongst instances of different closed constructed types. These rules apply regardless of whether the type of the static variable involves any type parameters or not.

For example:

```
class C<V>
{
   static int count = 0;

   public C() {
      count++;
   }

   public static int Count {
      get { return count; }
   }
}


class Application
{
   static void Main() {
      C<int> x1 = new C<int>();
      Console.WriteLine(C<int>.Count);    // Prints 1

      C<double> x2 = new C<double>();
      Console.WriteLine(C<int>.Count);    // Prints 1

      C<int> x3 = new C<int>();
      Console.WriteLine(C<int>.Count);    // Prints 2
   }
}
```

20.1.6 Static constructors in generic classes

Static constructors in generic classes are used to initialize static fields and perform other initialization for each different closed constructed type that is created from a particular generic class declaration. The type parameters of the generic type declaration are in scope and can be used within the body of the static constructor.

A new closed constructed class type is initialized the first time that either:

- An instance of the closed constructed type is created.
- Any of the static members of the closed constructed type are referenced.

To initialize a new closed constructed class type, first a new set of static fields (§20.1.5) for that particular closed constructed type is created. Each of the static fields is initialized to its default value (§5.2). Next, the static field initializers (§10.4.5.1) are executed for those static fields. Finally, the static constructor is executed.

Because the static constructor is executed exactly once for each closed constructed class type, it is a convenient place to enforce run-time checks on the type parameter that cannot be checked at compile-time via constraints (§20.6.6). For example, the following type uses a static constructor to enforce that the type parameter is a reference type:

```
class Gen<T>
{
   static Gen() {
      if ((object)T.default != null) {
         throw new ArgumentException("T must be a reference type");
      }
   }
}
```

20.1.7 Accessing protected members

Within a generic class declaration, access to inherited protected instance members is permitted through an instance of any class type constructed from the generic class. Specifically, the rules for accessing `protected` and `protected internal` instance members specified in §3.5.3 are augmented with the following rule for generics:

- Within a generic class `G`, access to an inherited protected instance member `M` using a *primary-expression* of the form `E.M` is permitted if the type of `E` is a class type constructed from `G` or a class type inherited from a class type constructed from `G`.

In the example

```
class C<T>
{
   protected T x;
}
class D<T>: C<T>
{
   static void F() {
      D<T> dt = new D<T>();
      D<int> di = new D<int>();
      D<string> ds = new D<string>();
      dt.x = T.default;
      di.x = 123;
      ds.x = "test";
   }
}
```

the three assignments to `x` are permitted because they all take place through instances of class types constructed from the generic type.

20.1.8 Overloading in generic classes

Methods, constructors, indexers, and operators within a generic class declaration can be overloaded; however, overloading is constrained so that ambiguities cannot occur within constructed classes. Two function members declared with the same names in the same generic class declaration must have parameter types such that no

closed constructed type could have two members with the same name and signature. When considering all possible closed constructed types, this rule includes type arguments that do not currently exist in the current program, but could be written. Type constraints on the type parameter are ignored for the purpose of this rule.

The following examples show overloads that are valid and invalid according to this rule:

```
interface I1<T> {...}

interface I2<T> {...}

class G1<U>
{
   long F1(U u);              // Invalid overload, G<int> would have two
   int F1(int i);             // members with the same signature

   void F2(U u1, U u2);       // Valid overload, no type argument for U
   void F2(int i, string s);  // could be int and string simultaneously

   void F3(I1<U> a);          // Valid overload
   void F3(I2<U> a);

   void F4(U a);              // Valid overload
   void F4(U[] a);
}

class G2<U,V>
{
   void F5(U u, V v);         // Invalid overload, G2<int,int> would have
   void F5(V v, U u);         // two members with the same signature

   void F6(U u, I1<V> v);     // Invalid overload, G2<I1<int>,int> would
   void F6(I1<V> v, U u);     // have two members with the same signature

   void F7(U u1, I1<V> v2);   // Valid overload, U cannot be V and I1<V>
   void F7(V v1, U u2);       // simultaneously

   void F8(ref U u);          // Invalid overload
   void F8(out V v);
}

class C1 {...}

class C2 {...}

class G3<U,V> where U: C1 where V: C2
{
   void F9(U u);              // Invalid overload, constraints on U and V
   void F9(V v);              // are ignored when checking overloads
}
```

20.1.9 Parameter array methods and type parameters

Type parameters may be used in the type of a parameter array. For example, given the declaration

```
class C<V>
{
   static void F(int x, int y, params V[] args);
}
```

the following invocations of the expanded form of the method:

```
C<int>.F(10, 20);
C<object>.F(10, 20, 30, 40);
C<string>.F(10, 20, "hello", "goodbye");
```

correspond exactly to:

```
C<int>.F(10, 20, new int[] {});
C<object>.F(10, 20, new object[] {30, 40});
C<string>.F(10, 20, new string[] {"hello", "goodbye"} );
```

20.1.10 Overriding and generic classes

Function members in generic classes can override function members in base classes, as usual. If the base class is a non-generic type or a closed constructed type, then any overriding function member cannot have constituent types that involve type parameters. However, if the base class is an open constructed type, then an overriding function member can use type parameters in its declaration. When determining the overridden base member, the members of the base classes must be determined by substituting type arguments, as described in §20.5.4. Once the members of the base classes are determined, the rules for overriding are the same as for non-generic classes.

The following example demonstrates how the overriding rules work in the presence of generics:

```
abstract class C<T>
{
   public virtual T F() {...}
   public virtual C<T> G() {...}
   public virtual void H(C<T> x) {...}
}
class D: C<string>
{
   public override string F() {...}         // Ok
   public override C<string> G() {...}      // Ok
   public override void H(C<T> x) {...}     // Error, should be C<string>
}
class E<T,U>: C<U>
{
   public override U F() {...}              // Ok
   public override C<U> G() {...}           // Ok
   public override void H(C<T> x) {...}     // Error, should be C<U>
}
```

20.1.11 Operators in generic classes

Generic class declarations may define operators, following the same rules as normal class declarations. The instance type (§20.1.2) of the class declaration must be used in the declaration of operators in a manner analogous to the normal rules for operators, as follows:

- A unary operator must take a single parameter of the instance type. The unary `++` and `--` operators must return the instance type.
- At least one of the parameters of a binary operator must be of the instance type.
- Either the parameter type or the return type of a conversion operator must be the instance type.

The following shows some examples of valid operator declarations in a generic class:

```
class X<T>
{
   public static X<T> operator ++(X<T> operand) {...}
   public static int operator *(X<T> op1, int op2) {...}
   public static explicit operator X<T>(T value) {...}
}
```

For a conversion operator that converts from a source type `S` to a target type `T`, when the rules specified in §10.9.3 are applied, any type parameters associated with `S` or `T` are considered to be unique types that have no inheritance relationship with other types, and any constrains on those type parameters are ignored.

In the example


```
class C<T> {...}
class D<T>: C<T>
{
   public static implicit operator C<int>(D<T> value) {...}    // Ok
   public static implicit operator C<T>(D<T> value) {...}      // Error
}
```

the first operator declaration is permitted because, for the purposes of §10.9.3, `T` and `int` are considered unique types with no relationship. However, the second operator is an error because `C<T>` is the base class of `D<T>`.

Given the above, it is possible to declare operators that, for some type arguments, specify conversions that already exist as pre-defined conversions. In the example

```
struct Nullable<T>
{
   public static implicit operator Nullable<T>(T value) {...}
   public static explicit operator T(Nullable<T> value) {...}
}
```

when type `object` is specified as a type argument for `T`, the second operator declares a conversion that already exists (an implicit, and therefore also an explicit, conversion exists from any type to type `object`).

In cases where a pre-defined conversion exists between two types, any user-defined conversions between those types are ignored. Specifically:

- If a pre-defined implicit conversion (§6.1) exists from type `S` to type `T`, all user-defined conversions (implicit or explicit) from `S` to `T` are ignored.
- If a pre-defined explicit conversion (§6.2) exists from type `S` to type `T`, any user-defined explicit conversions from `S` to `T` are ignored. However, user-defined implicit conversions from `S` to `T` are still considered.

For all types but `object`, the operators declared by the `Nullable<T>` type above do not conflict with pre-defined conversions. For example:

```
void F(int i, Nullable<int> n) {
   i = n;                       // Error
   i = (int)n;                  // User-defined explicit conversion
   n = i;                       // User-defined implicit conversion
   n = (Nullable<int>)i;        // User-defined implicit conversion
}
```

However, for type `object`, pre-defined conversions hide the user-defined conversions in all cases but one:

```
void F(object o, Nullable<object> n) {
   o = n;                       // Pre-defined boxing conversion
   o = (object)n;               // Pre-defined boxing conversion
   n = o;                       // User-defined implicit conversion
   n = (Nullable<object>)o;     // Pre-defined unboxing conversion
}
```

20.1.12 Nested types in generic classes

A generic class declaration can contain nested type declarations. The type parameters of the enclosing class may be used within the nested types. A nested type declaration may contain additional type parameters that apply only to the nested type.

Every type declaration contained within a generic class declaration is implicitly a generic type declaration. When writing a reference to a type nested within a generic type, the containing constructed type, including its type arguments, must be named. However, from within the outer class, the inner type can be used without qualification; the instance type of the outer class can be implicitly used when constructing the inner type. The

following example shows three different correct ways to refer to a constructed type created from `Inner`; the first two are equivalent:

```
class Outer<T>
{
   class Inner<U>
   {
      static void F(T t, U u) {...}
   }

   static void F(T t) {
      Outer<T>.Inner<string>.F(t, "abc");    // These two statements have
      Inner<string>.F(t, "abc");             // the same effect

      Outer<int>.Inner<string>.F(3, "abc");  // This type is different

      Outer.Inner<string>.F(t, "abc");       // Error, Outer needs type arg
   }
}
```

Although it is bad programming style, the type parameters in a nested type can hide a member or type parameter declared in the outer type:

```
class Outer<T>
{
   class Inner<T>    // Valid, hides Outer's T
   {
      public T t;    // Refers to Inner's T
   }
}
```

20.1.13 Application entry point

The application entry point method (§3.1) may not be in a generic class declaration.

20.2 Generic struct declarations

Like a class declaration, a `struct` declaration may optionally define type parameters:

struct-declaration:
attributes$_{opt}$ *struct-modifiers*$_{opt}$ `struct` *identifier* *type-parameter-list*$_{opt}$ *struct-interfaces*$_{opt}$
type-parameter-constraints-clauses$_{opt}$ *struct-body* `;`$_{opt}$

The rules for generic class declarations apply equally to generic struct declarations, except where the exceptions noted in §11.3 for *struct-declaration*s apply.

20.3 Generic interface declarations

Interfaces may also optionally define type parameters:

interface-declaration:
attributes$_{opt}$ *interface-modifiers*$_{opt}$ `interface` *identifier* *type-parameter-list*$_{opt}$
interface-base$_{opt}$ *type-parameter-constraints-clauses*$_{opt}$ *interface-body* `;`$_{opt}$

An interface that is declared with type parameters is a generic interface declaration. Except where noted, generic interface declarations follow the same rules as normal interface declarations.

Each type parameter in an interface declaration defines a name in the declaration space (§3.3) of that interface. The scope (§3.7) of a type parameter on an interface includes the *interface-base*, *type-parameter-constraints-clauses*, and *interface-body*. Within its scope, a type parameter can be used as a type. The same restrictions apply to type parameters on interfaces as apply to type parameter on classes (§20.1.1).

Methods within generic interfaces are subject to the same overload rules as methods within generic classes (§20.1.8).

20.3.1 Uniqueness of implemented interfaces

The interfaces implemented by a generic type declaration must remain unique for all possible constructed types. Without this rule, it would be impossible to determine the correct method to call for certain constructed types. For example, suppose a generic class declaration were permitted to be written as follows:

```
interface I<T>
{
   void F();
}

class X<U,V>: I<U>, I<V>               // Error: I<U> and I<V> conflict
{
   void I<U>.F() {...}
   void I<V>.F() {...}
}
```

Were this permitted, it would be impossible to determine which code to execute in the following case:

```
I<int> x = new X<int,int>();
x.F();
```

To determine if the interface list of a generic type declaration is valid, the following steps are performed:

- Let `L` be the list of interfaces directly specified in a generic class, struct, or interface declaration `C`.
- Add to `L` any base interfaces of the interfaces already in `L`.
- Remove any duplicates from `L`.
- If any possible constructed type created from `C` would, after type arguments are substituted into `L`, cause two interfaces in `L` to be indentical, then the declaration of `C` is invalid. Constraint declarations are not considered when determining all possible constructed types.

In the class declaration `X` above, the interface list `L` consists of `I<U>` and `I<V>`. The declaration is invalid because any constructed type with `U` and `V` being the same type would cause these two interfaces to be identical types.

20.3.2 Explicit interface member implementations

Explicit interface member implementations work with constructed interface types in essentially the same way as with simple interface types. As usual, an explicit interface member implementation must be qualified by an *interface-type* indicating which interface is being implemented. This type may be a simple interface or a constructed interface, as in the following example:

```
interface IList<T>
{
   T[] GetElements();
}
interface IDictionary<K,V>
{
   V this[K key];
   void Add(K key, V value);
}
class List<T>: IList<T>, IDictionary<int,T>
{
   T[] IList<T>.GetElements() {...}
   T IDictionary<int,T>.this[int index] {...}
```

```
   void IDictionary<int,T>.Add(int index, T value) {...}
}
```

20.4 Generic delegate declarations

A delegate declaration may include type parameters:

delegate-declaration:
attributes$_{opt}$ *delegate-modifiers*$_{opt}$ `delegate` *return-type* *identifier* *type-parameter-list*$_{opt}$ `(` *formal-parameter-list*$_{opt}$ `)` *type-parameter-constraints-clauses*$_{opt}$ `;`

A delegate that is declared with type parameters is a generic delegate declaration. A delegate declaration may not supply *type-parameter-constraints-clauses* (§20.7) unless it also supplies a *type-parameter-list*. Generic delegate declarations follow the same rules as normal delegate declarations, except where noted. Each type parameter in a generic delegate declaration defines a name in a special declaration space (§3.3) that is associated with that delegate declaration. The scope (§3.7) of a type parameter on a delegate declaration includes the *return-type*, *formal-parameter-list*, and *type-parameter-constraints-clauses*.

Like other generic type declarations, type arguments must be given to form a constructed delegate type. The parameter types and return type of a constructed delegate type are formed by substituting, for each type parameter in the delegate declaration, the corresponding type argument of the constructed delegate type. The resulting return type and parameter types are used for determining what methods are compatible (§15.1) with a constructed delegate type. For example:

```
delegate bool Predicate<T>(T value);

class X
{
   static bool F(int i) {...}

   static bool G(string s) {...}

   static void Main() {
      Predicate<int> p1 = F;
      Predicate<string> p2 = G;
   }
}
```

Note that the two assignments in the `Main` method above are equivalent to the following longer form:

```
static void Main() {
   Predicate<int> p1 = new Predicate<int>(F);
   Predicate<string> p2 = new Predicate<string>(G);
}
```

The shorter form is permitted because of method group conversions, which are described in §21.9.

20.5 Constructed types

A generic type declaration, by itself, does not denote a type. Instead, a generic type declaration is used as a "blueprint" to form many different types, by way of applying ***type arguments***. The type arguments are written within angle brackets (< and >) immediately following the name of the generic type declaration. A type that is named with at least one type argument is called a ***constructed type***. A constructed type can be used in most places in the language that a type name can appear.

type-name:
namespace-or-type-name

namespace-or-type-name:
identifier *type-argument-list*$_{opt}$
namespace-or-type-name `.` *identifier* *type-argument-list*$_{opt}$

Constructed types can also be used in expressions as simple names (§20.9.3) or when accessing a member (§20.9.4).

When a *namespace-or-type-name* is evaluated, only generic types with the correct number of type parameters are considered. Thus, it is possible to use the same identifier to identify different types, as long as the types have different numbers of type parameters and are declared in different namespaces. This is useful when mixing generic and non-generic classes in the same program:

```
namespace System.Collections
{
   class Queue {...}
}

namespace System.Collections.Generic
{
   class Queue<ElementType> {...}
}

namespace MyApplication
{
   using System.Collections;
   using System.Collections.Generic;

   class X
   {
      Queue q1;         // System.Collections.Queue
      Queue<int> q2;    // System.Collections.Generic.Queue
   }
}
```

The detailed rules for name lookup in these productions is described in §20.9. The resolution of ambiguities in these production is described in §20.6.5.

A *type-name* might identify a constructed type even though it doesn't specify type parameters directly. This can occur where a type is nested within a generic class declaration, and the instance type of the containing declaration is implicitly used for name lookup (§20.1.12):

```
class Outer<T>
{
   public class Inner {...}

   public Inner i;            // Type of i is Outer<T>.Inner
}
```

In unsafe code, a constructed type cannot be used as an *unmanaged-type* (§18.2).

20.5.1 Type arguments

Each argument in a type argument list is simply a *type*.

type-argument-list:
    `<` *type-arguments* `>`

type-arguments:
    type-argument
    type-arguments `,` *type-argument*

type-argument:
    type

Type arguments may in turn be constructed types or type parameters. In unsafe code (§18) a *type-argument* may not be a pointer type. Each type argument must satisfy any constraints on the corresponding type parameter (§20.7.1).

20.5.2 Open and closed types

All types can be classified as either ***open types*** or ***closed types***. An open type is a type that involves type parameters. More specifically:

- A type parameter defines an open type.
- An array type is an open type if and only if its element type is an open type.
- A constructed type is an open type if and only if one or more of its type arguments are an open type.

A closed type is a type that is not an open type.

At run-time, all of the code within a generic type declaration is executed in the context of a closed constructed type that was created by applying type arguments to the generic declaration. Each type parameter within the generic class is bound to a particular run-time type. The run-time processing of all statements and expressions always occurs with closed types, and open types occur only during compile-time processing.

Each closed constructed type has its own set of static variables, which are not shared with any other closed constructed types. Since an open type does not exist at run-time, there are no static variables associated with a open type. Two closed constructed types are the same type if they are constructed from the same type declaration, and corresponding type arguments are the same type.

20.5.3 Base classes and interfaces of a constructed type

A constructed class type has a direct base class, just like a simple class type. If the generic class declaration does not specify a base class, the base class is `object`. If a base class is specified in the generic class declaration, the base class of the constructed type is obtained by substituting, for each *type-parameter* in the base class declaration, the corresponding *type-argument* of the constructed type. Given the generic class declarations

```
class B<U,V> {...}
class G<T>: B<string,T[]> {...}
```

the base class of the constructed type `G<int>` would be `B<string,int[]>`.

Similarly, constructed class, struct, and interface types have a set of explicit base interfaces. The explicit base interfaces are formed by taking the explicit base interface declarations on the generic type declaration, and substituting, for each *type-parameter* in the base interface declaration, the corresponding *type-argument* of the constructed type.

The set of all base classes and base interfaces for a type is formed, as usual, by recursively getting the base classes and interfaces of the immediate base classes and interfaces. For example, given the generic class declarations:

```
class A {...}
class B<T>: A {...}
class C<T>: B<IComparable<T>> {...}
class D<T>: C<T[]> {...}
```

the base classes of `D<int>` are `C<int[]>`, `B<IComparable<int[]>>`, `A`, and `object`.

20.5.4 Members of a constructed type

The non-inherited members of a constructed type are obtained by substituting, for each *type-parameter* in the member declaration, the corresponding *type-argument* of the constructed type.

For example, given the generic class declaration

```
class Gen<T,U>
{
   public T[,] a;
   public void G(int i, T t, Gen<U,T> gt) {...}
   public U Prop { get {...} set {...} }
   public int H(double d) {...}
}
```

the constructed type `Gen<int[],IComparable<string>>` has the following members:

```
public int[,][] a;
public void G(int i, int[] t, Gen<IComparable<string>,int[]> gt) {...}
public IComparable<string> Prop { get {...} set {...} }
public int H(double d) {...}
```

Note that the substitution process is based on the semantic meaning of type declarations, and is not simply textual substitution. The type of the member `a` in the generic class declaration `Gen` is "two-dimensional array of `T`", so the type of the member `a` in the instantiated type above is "two-dimensional array of one-dimensional array of `int`", or `int[,][]`.

The inherited members of a constructed type are obtained in a similar way. First, all the members of the immediate base class are determined. If the base class is itself a constructed type, this may involved a recursive application of the current rule. Then, each of the inherited members is transformed by substituting, for each *type-parameter* in the member declaration, the corresponding *type-argument* of the constructed type.

```
class B<U>
{
   public U F(long index) {...}
}
class D<T>: B<T[]>
{
   public T G(string s) {...}
}
```

In the above example, the constructed type `D<int>` has a non-inherited member `public int G(string s)` obtained by substituting the type argument `int` for the type parameter `T`. `D<int>` also has an inherited member from the class declaration `B`. This inherited member is determined by first determining the members of the constructed type `B<T[]>` by substituting `T[]` for `U`, yielding `public T[] F(long index)`. Then, the type argument `int` is substituted for the type parameter `T`, yielding the inherited member `public int[] F(long index)`.

20.5.5 Accessibility of a constructed type

A constructed type $C<T_1, ..., T_N>$ is accessible when all its parts C, T_1, ..., T_N are accessible. For instance, if the generic type name `C` is `public` and all of the *type-arguments* T_1, ...,T_N are accessible as `public`, then the constructed type is accessible as `public`, but if either the *type-name* or one of the *type-arguments* has accessibility `private` then the accessibility of the constructed type is `private`. If one *type-argument* has accessibility `protected`, and another has accessibility `internal`, then the constructed type is accessible only in this class and its subclasses in this assembly.

More precisely, the accessibility domain for a constructed type is the intersection of the accessibility domains of its constituent parts. Thus if a method has a return type or argument type that is a constructed type where one constituent part is `private`, then the method must have an accessibility domain that is `private`; see §3.5.

20.5.6 Conversions

Constructed types follow the same conversion rules (§6) as do non-generic types. When applying these rules, the base classes and interfaces of constructed types must be determined as described in §20.5.3.

No special conversions exist between constructed reference types other than those described in §6. In particular, unlike array types, constructed reference types do not exhibit "co-variant" conversions. This means that a type `List<B>` has no conversion (either implicit or explicit) to `List<A>` even if `B` is derived from `A`. Likewise, no conversion exists from `List<B>` to `List<object>`.

The rationale for this is simple: if a conversion to `List<A>` is permitted, then apparently one can store values of type `A` into the list. But this would break the invariant that every object in a list of type `List<B>` is always a value of type `B`, or else unexpected failures may occur when assigning into collection classes.

The behavior of conversions and runtime type checks is illustrated below:

```
class A {...}

class B: A {...}

class Collection {...}

class List<T>: Collection {...}

class Test
{
   void F() {
      List<A> listA = new List<A>();
      List<B> listB = new List<B>();

      Collection c1 = listA;           // Ok, List<A> is a Collection
      Collection c2 = listB;           // Ok, List<B> is a Collection

      List<A> a1 = listB;              // Error, no implicit conversion
      List<A> a2 = (List<A>)listB;     // Error, no explicit conversion
   }
}
```

20.5.7 The System.Nullable<T> type

The `System.Nullable<T>` generic struct type defined in the .NET Base Class Library represents a value of type `T` that may be null. The `System.Nullable<T>` type is useful in a variety of situations, such as to denote nullable columns in a database table or optional attributes in an XML element.

An implicit conversion exists from the null type to any type constructed from `System.Nullable<T>`. The result of such a conversion is the default value of `System.Nullable<T>`. In other words, writing

```
Nullable<int> x = null;
Nullable<string> y = null;
```

is the same as writing

```
Nullable<int> x = Nullable<int>.default;
Nullable<string> y = Nullable<string>.default;
```

20.5.8 Using alias directives

Using aliases may name a closed constructed type, but may not name a generic type declaration without supplying type arguments. For example:

```
namespace N1
{
   class A<T>
   {
      class B {}
   }
```

```
   class C {}
}
namespace N2
{
   using W = N1.A;         // Error, cannot name generic type
   using X = N1.A.B;       // Error, cannot name generic type
   using Y = N1.A<int>;    // Ok, can name closed constructed type
   using Z = N1.C;         // Ok
}
```

20.5.9 Attributes

An open type may not be used anywhere inside an attribute. A closed constructed type can be used as the argument to an attribute, but cannot be used as the *attribute-name*, because `System.Attribute` cannot be the base type of a generic class declaration.

```
class A: Attribute
{
   public A(Type t) {...}
}
class B<T>: Attribute {}           // Error, cannot use Attribute as base
class List<T>
{
   [A(typeof(T))] T t;             // Error, open type in attribute
}
class X
{
   [A(typeof(List<int>))] int x;   // Ok, closed constructed type
   [B<int>] int y;                 // Error, invalid attribute name
}
```

20.6 Generic methods

A generic method is a method that is generic with respect to certain types. A generic method declaration names, in addition to normal parameters, a set of type parameters which are provided when using the method. Generic methods may be declared inside class, struct, or interface declarations, which may themselves be either generic or non-generic. If a generic method is declared inside a generic type declaration, the body of the method can refer to both the type parameters of the method, and the type parameters of the containing declaration.

class-member-declaration:
    …
    generic-method-declaration

struct-member-declaration:
    …
    generic-method-declaration

interface-member-declaration:
    …
    interface-generic-method-declaration

Generic methods are declared by placing a type parameter list following the name of the method:

generic-method-declaration:
    generic-method-header method-body

generic-method-header:
attributes$_{opt}$ *method-modifiers*$_{opt}$ *return-type* *member-name* *type-parameter-list*
(*formal-parameter-list*$_{opt}$) *type-parameter-constraints-clauses*$_{opt}$

interface-generic-method-declaration:
attributes$_{opt}$ `new`$_{opt}$ *return-type* *identifier* *type-parameter-list*
(*formal-parameter-list*$_{opt}$) *type-parameter-constraints-clauses*$_{opt}$;

The *type-parameter-list* and *type-parameter-constraints-clauses* have the same syntax and function as in a generic type declaration. The *method-type-parameters* are in scope throughout the *generic-method-declaration*, and may be used to form types throughout that scope including the *return-type*, the *method-body*, and the *type-parameter-constraints-clauses* but excluding the *attributes*.

The name of a method type parameter cannot be the same as the name of an ordinary parameter to the same method.

The following example finds the first element in an array, if any, that satisfies the given test delegate. Generic delegates are described in §20.4.

```
public delegate bool Test<T>(T item);

public class Finder
{
   public static T Find<T>(T[] items, Test<T> test) {
      foreach (T item in items) {
         if (test(item)) return item;
      }
      throw new InvalidOperationException("Item not found");
   }
}
```

A generic method may not be declared `extern`. All other method modifiers are valid on a generic method.

20.6.1 Generic method signatures

For the purposes of signature comparisons any *type-parameter-constraints-clauses* are ignored, as are the names of the *method-type-parameter*s, but the number of generic type parameters is relevant, as are the ordinal positions of type-parameters in left-to-right ordering. The following example shows how method signatures are affected by this rule:

```
class A {}

class B {}

interface IX
{
   T F1<T>(T[] a, int i);     // Error, both declarations have the same
   void F1<U>(U[] a, int i);  // signature because return type and type
                              // parameter names are not significant

   void F2<T>(int x);         // Ok, the number of type parameters is part
   void F2(int x);            // of the signature

   void F3<T>(T t) where T: A;   // Error, constraints are not
   void F3<T>(T t) where T: B;   // considered in signatures
}
```

Overloading of generic methods is further constrained by a rule similar to that which governs overloaded methods in a generic type declaration (20.1.8). Two generic methods declared with the same names and same number of type arguments must have parameter types such that no list of closed type arguments, when applied to both methods in the same order, yield two methods with the same signature. Constraints are not considered for the purposes of this rule. For example:

```
class X<T>
{
   void F<U>(T t, U u) {...}  // Error, X<int>.F<int> yields two methods
   void F<U>(U u, T t) {...}  // with the same signature
}
```

20.6.2 Virtual generic methods

Generic methods can be declared using the `abstract`, `virtual`, and `override` modifiers. The signature matching rules described in §20.6.1 are used when matching methods for overriding or interface implementation. When a generic method overrides a generic method declared in a base class, or implements a method in a base interface, the constraints given for each method type parameter must be the same in both declarations, where method type parameters are identified by ordinal positions, left to right.

```
abstract class Base
{
   public abstract T F<T,U>(T t, U u);

   public abstract T G<T>(T t) where T: IComparable;
}

class Derived: Base
{
   public override X F<X,Y>(X x, Y y) {...}     // Ok

   public override T G<T>(T t) {...}            // Error
}
```

The override of `F` is correct because type parameter names are permitted to differ. The override of `G` is incorrect because the given type parameter constraints (in this case none) do not match those of the method being overridden.

20.6.3 Calling generic methods

A generic method invocation may explicitly specify a type argument list, or it may omit the type argument list and rely on type inference to determine the type arguments. The exact compile-time processing of a method invocation, including a generic method invocation, is described in §20.9.5. When a generic method is invoked without a type argument list, type inference takes place as described in §20.6.4.

The following example shows how overload resolution occurs after type inference and after type arguments are substituted into the parameter list:

```
class Test
{
   static void F<T>(int x, T y) {
      Console.WriteLine("one");
   }

   static void F<T>(T x, long y) {
      Console.WriteLine("two");
   }

   static void Main() {
      F<int>(5, 324);       // Ok, prints "one"
      F<byte>(5, 324);      // Ok, prints "two"
      F<double>(5, 324);    // Error, ambiguous

      F(5, 324);            // Ok, prints "one"
      F(5, 324L);           // Error, ambiguous
   }
}
```

20.6.4 Inference of type arguments

When a generic method is called without specifying type arguments, a ***type inference*** process attempts to infer type arguments for the call. The presence of type inference allows a more convenient syntax to be used for calling a generic method, and allows the programmer to avoid specifying redundant type information. For example, given the method declaration:

```
class Util
{
   static Random rand = new Random();

   static public T Choose<T>(T first, T second) {
      return (rand.Next(2) == 0)? first: second;
   }
}
```

it is possible to invoke the `Choose` method without explicitly specifying a type argument:

```
int i = Util.Choose(5, 213);              // Calls Choose<int>
string s = Util.Choose("foo", "bar");     // Calls Choose<string>
```

Through type inference, the type arguments `int` and `string` are determined from the arguments to the method.

Type inference occurs as part of the compile-time processing of a method invocation (§20.9.5) and takes place before the overload resolution step of the invocation. When a particular method group is specified in a method invocation, and no type arguments are specified as part of the method invocation, type inference is applied to each generic method in the method group. If type inference succeeds, then the inferred type arguments are used to determine the types of arguments for subsequent overload resolution. If overload resolution chooses a generic method as the one to invoke, then the inferred type arguments are used as the actual type arguments for the invocation. If type inference for a particular method fails, that method does not participate in overload resolution. The failure of type inference, in and of itself, does not cause a compile-time error. However, it often leads to a compile-time error when overload resolution then fails to find any applicable methods.

If the supplied number of arguments is different than the number of parameters in the method, then inference immediately fails. Otherwise, type inference first occurs independently for each regular argument that is supplied to the method. Assume this argument has type `A`, and the corresponding parameter has type `P`. Type inferences are produced by relating the types `A` and `P` according to the following steps:

- Nothing is inferred from the argument (but type inference succeeds) if any of the following are true:
  - `P` does not involve any method type parameters.
  - The argument is the `null` literal.
  - The argument is an anonymous method.
  - The argument is a method group.
- If `P` is an array type and `A` is an array type of the same rank, then replace `A` and `P` respectively with the element types of `A` and `P` and repeat this step.
- If `P` is an array type and `A` is not an array type of the same rank, then type inference fails for the generic method.
- If `P` is a method type parameter, then type inference succeeds for this argument, and `A` is the type inferred for that type parameter.
- Otherwise, `P` must be a constructed type. If, for each method type parameter M_X that occurs in `P`, exactly one type T_X can be determined such that replacing each M_X with each T_X produces a type to which `A` is convertible by a standard implicit conversion, then inferencing succeeds for this argument, and each T_X is the type inferred for each M_X. Method type parameter constraints, if any, are ignored for the purpose of type inference.

If, for a given M_X, no T_X exists or more than one T_X exists, then type inference fails for the generic method (a situation where more than one T_X exists can only occur if P is a generic interface type and A implements multiple constructed versions of that interface).

If all of the method arguments are processed successfully by the above algorithm, then all inferences that were produced from the arguments are pooled. This pooled set of inferences must have the following properties:

- Each type parameter of the method must have had a type argument inferred for it. In short, the set of inferences must be ***complete***.
- If a type parameter occurred more than once, then all of the inferences for that type parameter must infer the same type argument. In short, the set of inferences must be ***consistent***.

If a complete and consistent set of inferred type arguments is found, then type inference is said to have succeeded for the given generic method and argument list.

If the generic method was declared with a parameter array (§10.5.1.4), then type inference is first performed against the method in its normal form. If type inference succeeds, and the resultant method is applicable, then the method is eligible for overload resolution in its normal form. Otherwise, type inference is performed against the method in its expanded form (§7.4.2.1).

20.6.5 Grammar ambiguities

The productions for *simple-name* and *member-access* in §20.6.3 can give rise to ambiguities in the grammar for expressions. For example, the statement:

```
F(G<A,B>(7));
```

could be interpreted as a call to F with two arguments, G < A and B > (7). Alternatively, it could be interpreted as a call to F with one argument, which is a call to a generic method G with two type arguments and one regular argument.

If an expression could be parsed in two different valid ways, where > can be either interpreted as all or part of an operator, or as ending a *type-argument-list*, the token immediately following the > is examined. If it is one of

```
(  )  ]  >  :  ;  ,  .  ?
```

then the > is interpreted as the end of a *type-argument-list*. Otherwise, the > is interpreted as an operator.

20.6.6 Using a generic method with a delegate

An instance of a delegate can be created that refers to a generic method declaration. The exact compile-time processing of a delegate creation expression, including a delegate creation expression that refers to a generic method, is described in §20.9.6.

The type arguments used when invoking a generic method through a delegate are determined when the delegate is instantiated. The type arguments can be given explicitly via a type-argument-list, or determined by type inference (§20.6.4). If type inference is used, the parameter types of the delegate are used as argument types in the inference process. The return type of the delegate is *not* used for inference. The following example shows both ways of supplying a type argument to a delegate instantiation expression:

```
delegate int D(string s, int i);
delegate int E();
class X
{
   public static T F<T>(string s, T t) {...}
   public static T G<T>() {...}
```

```
   static void Main() {
      D d1 = new D(F<int>);      // Ok, type argument given explicitly
      D d2 = new D(F);           // Ok, int inferred as type argument

      E e1 = new E(G<int>);      // Ok, type argument given explicitly
      E e2 = new E(G);           // Error, cannot infer from return type
   }
}
```

In the above example, a non-generic delegate type was instantiated using a generic method. It is also possible to create an instance of a constructed delegate type (§20.4) using a generic method. In all cases, type arguments are given or inferred when the delegate instance is created, and a *type-argument-list* may not be supplied when a delegate is invoked (§15.3).

20.6.7 No generic properties, events, indexers, or operators

Properties, events, indexers, and operators may not themselves have type parameters (although they can occur in generic classes, and use the type parameters from an enclosing class). If a property-like construct is required that must itself be generic, a generic method must be used instead.

20.7 Constraints

Generic type and method declarations can optionally specify type parameter constraints by including *type-parameter-constraints-clauses* in the declaration.

type-parameter-constraints-clauses:
    type-parameter-constraints-clause
    type-parameter-constraints-clauses *type-parameter-constraints-clause*

type-parameter-constraints-clause:
    `where` *type-parameter* `:` *type-parameter-constraints*

type-parameter-constraints:
    class-constraint
    interface-constraints
    constructor-constraint
    class-constraint `,` *interface-constraints*
    class-constraint `,` *constructor-constraint*
    interface-constraints `,` *constructor-constraint*
    class-constraint `,` *interface-constraints* `,` *constructor-constraint*

class-constraint:
    class-type

interface-constraints:
    interface-constraint
    interface-constraints `,` *interface-constraint*

interface-constraint:
    interface-type

constructor-constraint:
    `new ( )`

Each type parameter constraints clause consists of the token `where`, followed by the name of a type parameter, followed by a colon and the list of constraints for that type parameter. There can be only one `where` clause for each type parameter, but the `where` clauses may be listed in any order. Similar to the `get` and `set` tokens in a property accessor, the `where` token is not a keyword.

The list of constraints given in a `where` clause may include any of the following components, in this order: a single class constraint, one or more interface constraints, and the constructor constraint `new()`.

If a constraint is a class type or an interface type, that type specifies a minimal "base type" that every type argument used for that type parameter must support. Whenever a constructed type or generic method is used, the type argument is checked against the constraints on the type parameter at compile-time. The type argument supplied must derive from or implement all of the constraints given for that type parameter.

The type specified as a *class-constraint* must satisfy the following rules:

- The type must be a class type.
- The type must not be `sealed`.
- The type must not be one of the following special types: `System.Array`, `System.Delegate`, `System.Enum`, or `System.ValueType`.
- The type must not be `object`. Since all types derive from `object`, such a constraint would have no effect if it were permitted.
- At most one constraint for a given type parameter can be a class type.

The type specified as a *interface-constraint* must satisfy the following rules:

- The type must be an interface type.
- The same type may not be specified more than once in a given `where` clause.

In either case, the constraint may involve any of the type parameters of the associated type or method declaration as part of a constructed type, and may involve the type being declared, but the constraint may not be a type parameter alone.

Any class or interface type specified as a type parameter constraint must be at least as accessible (§10.5.4) as the generic type or method being declared.

If the `where` clause for a type parameter includes a constructor constraint of the form `new()`, it is possible to use the `new` operator to create instances of the type (§20.8.2). Any type argument used for a type parameter with a constructor constraint must have an parameterless constructor (see §20.7.1 for details).

The following are examples of possible constraints:

```
interface IPrintable
{
   void Print();
}

interface IComparable<T>
{
   int CompareTo(T value);
}

interface IKeyProvider<T>
{
   T GetKey();
}

class Printer<T> where T: IPrintable {...}

class SortedList<T> where T: IComparable<T> {...}
```

```
class Dictionary<K,V>
   where K: IComparable<K>
   where V: IPrintable, IKeyProvider<K>, new()
{
   ...
}
```

The following example is in error because it attempts to use a type parameter directly as a constraint:

```
class Extend<T,U> where U: T {...}  // Error
```

Values of a constrained type parameter type can be used to access the instance members implied by the constraints. In the example

```
interface IPrintable
{
   void Print();
}
class Printer<T> where T: IPrintable
{
   void PrintOne(T x) {
      x.Print();
   }
}
```

the methods of `IPrintable` can be invoked directly on `x` because `T` is constrained to always implement `IPrintable`.

20.7.1 Satisfying constraints

Whenever a constructed type is used or a generic method is referenced, the supplied type arguments are checked against the type parameter constraints declared on the generic type or method. For each `where` clause, the type argument `A` that corresponds to the named type parameter is checked against each constraint as follows:

- If the constraint is a class type or an interface type, let `C` represent that constraint with the supplied type arguments substituted for any type parameters that appear in the constraint. To satisfy the constraint, it must be the case that type `A` is convertible to type `C` by one of the following:
  - An identity conversion (§6.1.1)
  - An implicit reference conversion (§6.1.4)
  - A boxing conversion (§6.1.5)
  - An implicit conversion from a type parameter `A` to `C` (§20.7.4).
- If the constraint is `new()`, the type argument `A` must not be `abstract` and must have a parameterless constructor that is at least as accessible (§3.5.4) as the containing type. This is satisfied if either:
  - `A` is a value type, since all value types have a public default constructor (§4.1.2), or
  - `A` is a class that is not `abstract`, `A` contains an explicitly declared constructor with no parameters, and that constructor is at least as accessible as `A`.
  - `A` is not `abstract` and has a default constructor (§10.10.4).

A compile time error occurs if one or more of a type parameter's constraints are not satisfied by the given type arguments.

Since type parameters are not inherited, constraints are never inherited either. In the example below, `D` must specify a constraint on its type parameter `T`, so that `T` satisfies the constraint imposed by the base class `B<T>`. In contrast, class `E` need not specify a constraint, because `List<T>` implements `IEnumerable` for any `T`.