C

Version 2.0 Specification

July 2003

Table of Contents

19. Introduction to C# 2.0 1
19.1 Generics 1
19.1.1 Why generics? 1
19.1.2 Creating and using generics 2
19.1.3 Generic type instantiations 3
19.1.4 Constraints 3
19.1.5 Generic methods 5
19.2 Anonymous methods 5
19.2.1 Method group conversions 8
19.3 Iterators 8
19.4 Partial types 11

20. Generics 13
20.1 Generic class declarations 13
20.1.1 Type parameters 13
20.1.2 The instance type 14
20.1.3 Base specification 15
20.1.4 Members of generic classes 15
20.1.5 Static fields in generic classes 16
20.1.6 Static constructors in generic classes 16
20.1.7 Accessing protected members 17
20.1.8 Overloading in generic classes 17
20.1.9 Parameter array methods and type parameters 18
20.1.10 Overriding and generic classes 19
20.1.11 Operators in generic classes 19
20.1.12 Nested types in generic classes 20
20.1.13 Application entry point 21
20.2 Generic struct declarations 21
20.3 Generic interface declarations 21
20.3.1 Uniqueness of implemented interfaces 22
20.3.2 Explicit interface member implementations 22
20.4 Generic delegate declarations 23
20.5 Constructed types 23
20.5.1 Type arguments 24
20.5.2 Open and closed types 25
20.5.3 Base classes and interfaces of a constructed type 25
20.5.4 Members of a constructed type 25
20.5.5 Accessibility of a constructed type 26
20.5.6 Conversions 27
20.5.7 The System.Nullable<T> type 27
20.5.8 Using alias directives 27
20.5.9 Attributes 28
20.6 Generic methods 28
20.6.1 Generic method signatures 29
20.6.2 Virtual generic methods 30
20.6.3 Calling generic methods 30
20.6.4 Inference of type arguments 31
20.6.5 Grammar ambiguities 32
20.6.6 Using a generic method with a delegate 32

20.6.7 No generic properties, events, indexers, or operators 33
20.7 Constraints 33
20.7.1 Satisfying constraints 35
20.7.2 Member lookup on type parameters 36
20.7.3 Type parameters and boxing 36
20.7.4 Conversions involving type parameters 37
20.8 Expressions and Statements 39
20.8.1 Default value expression 39
20.8.2 Object creation expressions 39
20.8.3 The typeof operator 39
20.8.4 Reference equality operators 40
20.8.5 The is operator 40
20.8.6 The as operator 40
20.8.7 Exception statements 41
20.8.8 The lock statement 41
20.8.9 The using statement 41
20.8.10 The foreach statement 41
20.9 Revised lookup rules 42
20.9.1 Namespace and type names 42
20.9.2 Member lookup 43
20.9.3 Simple names 44
20.9.4 Member access 45
20.9.5 Method invocations 47
20.9.6 Delegate creation expressions 48
20.10 Right-shift grammar changes 49

21. Anonymous methods 51

21.1 Anonymous method expressions 51
21.2 Anonymous method signatures 51
21.3 Anonymous method conversions 51
21.3.1 Delegate creation expression 53
21.4 Anonymous method blocks 53
21.5 Outer variables 53
21.5.1 Captured outer variables 54
21.5.2 Instantiation of local variables 54
21.6 Anonymous method evaluation 56
21.7 Delegate instance equality 57
21.8 Definite assignment 57
21.9 Method group conversions 58
21.10 Implementation example 59

22. Iterators 63

22.1 Iterator blocks 63
22.1.1 Enumerator interfaces 63
22.1.2 Enumerable interfaces 63
22.1.3 Yield type 63
22.1.4 This access 64
22.2 Enumerator objects 64
22.2.1 The MoveNext method 64
22.2.2 The Current property 65
22.2.3 The Dispose method 66

22.3 Enumerable objects 66
22.3.1 The GetEnumerator method 66
22.4 The yield statement 67
22.4.1 Definite assignment 68
22.5 Implementation example 68

23. Partial Types 73

23.1 Partial declarations 73
23.1.1 Attributes 73
23.1.2 Modifiers 74
23.1.3 Type parameters and constraints 74
23.1.4 Base class 74
23.1.5 Base interfaces 75
23.1.6 Members 75
23.2 Name binding 76

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 requi472.29 Tm (e)Tj 0 1 392.04 5(472.2.29 Tr

```
            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
```

```
   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);
```

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 {}
   }
```

tac t ..OO

```
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`, `vir`

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>` implemes

```
class B<T> where T: IEnumerable {...}
class D<T>: B<T> where T: IEnumerable {...}
class E<T>: B<List<T>> {...}
```

20.7.2 Member lookup on type parameters

The results of member lookup in a type given by a type parameter `T` depends on the constraints, if any, specified for `T`. If `T` has no constraints, or only the `new()` constraint, then member lookup on `T` returns the same set of members as member lookup on `object`. Otherwise, the first stage of member lookup (§20.9.2) considers all the members in each of the types that are constraints for `T`. After performing the first stage of member lookup for each of the type constraints of `T`, the results are combined, and then hidden members are removed from the combined results.

Before the advent of generics, member lookup always returned either a set of members declared solely in classes, or a set of members declared solely in interfaces and possibly the type `object`. Member lookup on type parameters changes this somewhat. When a type parameter has both a class constraint and one or more interface constraints, member lookup can return a set of members, some of which were declared in the class, and others of which were declared in an interface. The following additional rules handle this case.

- During member lookup (§20.9.2), members declared in a class other than `object` hide members declared in interfaces.
- During overload resolution of methods (§7.5.5.1) and indexers (§7.5.6.2), if any applicable member was declared in a class other than `object`, all members declared in an interface are removed from the set of considered members.

These rules only have effect when doing binding on a type parameter with both a class constraint and an interface constraint. Informally, members defined in a class constraint are always preferred over members in an interface constraint.

20.7.3 Type parameters and boxing

When a struct type overrides a virtual method inherited from `System.Object` (`Equals`, `GetHashCode`, or `ToString`), invocations of the virtual method through an instance of the struct type doesn't cause boxing to occur. This is true even when the struct is used as a type parameter and the invocation occurs through an instance of the type parameter type. For example:

```
using System;

struct Counter
{
   int value;

   public override string ToString() {
      value++;
      return value.ToString();
   }
}

class Program
{
   static void Test<T>() where T: new() {
      T x = new T();
      Console.WriteLine(x.ToString());
      Console.WriteLine(x.ToString());
      Console.WriteLine(x.ToString());
   }
```

```
   static void Main() {
      Test<Counter>();
   }
}
```

The output of the program is:

```
1
2
3
```

Although it is never recommended for `ToString` to have side effects, the example demonstrates that no boxing occurred for the three invocations of `x.ToString()`.

Boxing never implicitly occurs when accessing a member on a constrained type parameter. For example, suppose an interface `ICounter` contains a method `Increment` which can be used to modify a value. If `ICounter` is used as a constraint, the implementation of the `Increment` method is called with a reference to the variable that `Increment` was called on, never a boxed copy:

```
using System;

interface ICounter
{
   void Increment();
}

struct Counter: ICounter
{
   int value;

   public override string ToString() {
      return value.ToString();
   }

   void ICounter.Increment() {
      value++;
   }
}

class Program
{
   static void Test<T>() where T: new(), ICounter {
      T x = new T();
      Console.WriteLine(x);
      x.Increment();               // Modify x
      Console.WriteLine(x);
      ((ICounter)x).Increment();   // Modify boxed copy of x
      Console.WriteLine(x);
   }

   static void Main() {
      Test<Counter>();
   }
}
```

The first call to `Increment` modifies the value in the variable `x`. This is not equivalent to the second call to `Increment`, which modifies the value in a boxed copy of `x`. Thus, the output of the program is:

```
0
1
1
```

20.7.4 Conversions involving type parameters

The conversions that are allowed on a type parameter `T` depend on the constraints specified for `T`. All type parameters, constrained or not, have the following conversions.

- An implicit identity conversion from `T` to `T`.
- An implicit conversion from `T` to `object`. At run-time, if `T` is a value type, this is executed as a boxing conversion. Otherwise, it is executed as an implicit reference conversion.
- An explicit conversion from `object` to `T`. At run-time, if `T` is a value type, this is executed as an unboxing conversion. Otherwise, it is executed as an explicit reference conversion.
- An explicit conversion from `T` to any interface type. At run-time, if `T` is a value type, this is executed as a boxing conversion. Otherwise, it is executed as an explicit reference conversion.
- An explicit conversion from any interface type to `T`. At run-time, if `T` is a value type, this is executed as an unboxing conversion. Otherwise, it is executed as an explicit reference conversion.

If the type parameter `T` has the interface type `I` specified as a constraint, the following additional conversions exist:

- An implicit conversion from `T` to `I`, and from `T` to any base interface type of `I`. At run-time, if `T` is a value type, this is executed as a boxing conversion. Otherwise, it is executed as an implicit reference conversion.

If the type parameter `T` has the class type `C` specified as a constraint, the following additional conversions exist:

- An implicit reference conversion from `T` to `C`, from `T` to any class `C` is derived from, and from `T` to any interface `C` implements.
- An explicit reference conversion from `C` to `T`, from any class `C` is derived from to `T`, and from any interface `C` implements to `T`.
- An implicit user-defined conversion from `T` to `A`, if an implicit user-defined conversion exists from `C` to `A`.
- An explicit user-defined conversion from `A` to `T`, if an explicit user-defined conversion exists from `A` to `C`.
- An implicit reference conversion from the null type to `T`.

An array type with element type `T` has the usual conversions to and from `object` and `System.Array` (§6.1.4, §6.2.3). If `T` has a class type `C` specified as a constraint, then additionally:

- An implicit reference conversion exists from an array type A_T with element type `T` to an array type A_U with element type `U`, and an explicit reference conversion exist from A_U to A_T, if both the following are true:
  - A_T and A_U have the same number of dimensions.
  - `U` is one of: `C`, a class `C` is derived from, an interface `C` implements, an interface `I` that is specified as a constraint on `T`, or a base interface of `I`.

The above rules do not permit a direct explicit conversion from an unconstrained type parametes

```
class X<T>
{
   public static long F(T t) {
      return (long) (object) t;     // OK, conversions permitted
   }
}
```

20.8 Expressions and Statements

The operation of some expressions and statements is modified with generics. This section details those changes.

20.8.1 Default value expression

A default value expression is used to obtain the default value (§5.2) of a type. Typically a default value expression is used for type parameters, since it may not be known if the type parameter is a value type or a reference type. (No conversion exists from the `null` literal to a type parameter.)

primary-no-array-creation-expression:
    …
    default-value-expression

default-value-expression:
    primary-expression `.` `default`
    predefined-type `.` `default`

If a *primary-expression* is used in a *default-value-expression*, and the *primary-expression* is not classified as a type, then a compile-time error occurs. However, the rule described in §7.5.4.1 also applies to a construct of the form `E.default`.

If the left hand side of a *default-value-expression* evaluates at run-time to a reference type, the result is `null` converted to that type. If the left hand side of a *default-value-expression* evaluates at run-time to a value type, the result is the *value-type*'s default value (§4.1.2).

A *default-value-expression* is a constant expression (§7.15) if the type is a reference type or a type parameter that has a class constraint. In addition, a *default-value-expression* is a constant expression if the type is one of the following value types: `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, or `bool`.

20.8.2 Object creation expressions

The type of an object creation expression can be a type parameter. When a type parameter is specified as the type in an object creation expression, both of the following conditions must hold, or a compile-time error occurs:

- The argument list must be omitted.
- A constructor constraint of the form `new()` must have been specified for the type parameter.

Execution of the object creation expression occurs by creating an instance of the run-time type that the type parameter has been bound to, and invoking the default constructor of that type. The run-time type may be a reference type or a value type.

20.8.3 The typeof operator

The `typeof` operator can be used on a *type-parameter*. The result is the `System.Type` object for the run-time type that was bound to the type-parameter. The `typeof` operator can also be used on a constructed type.

```
class X<T>
{
   public static void PrintTypes() {
      Console.WriteLine(typeof(T).FullName);
      Console.WriteLine(typeof(X<X<T>>).FullName);
   }
}

class M
{
   static void Main() {
      X<int>.PrintTypes();
   }
}
```

The above program will print:

```
System.Int32
X<X<System.Int32>>
```

The `typeof` operator cannot be used with the name of a generic type declaration without specifying the type arguments:

```
class X<T> {...}

class M
{
   static void Main() {
      Type t = typeof(X);     // Error, X requires type arguments
   }
}
```

20.8.4 Reference equality operators

The reference type equality operators (§7.9.6) may be used to compare values of a type parameter `T` if `T` is constrained by a class constraint.

The use of the reference type equality operators is slightly relaxed to allow one argument to be of a type parameter `T` and the other argument to be `null`, even if `T` has no class constraint. At run-time, if `T` is a value type, the result of the comparison is `false`.

The following example checks whether an argument of an unconstrained type parameter type is `null`.

```
class C<T>
{
   void F(T x) {
      if (x == null) throw new ArgumentNullException();
      ...
   }
}
```

The `x == null` construct is permitted even though `T` could represent a value type, and the result is simply defined to be `false` when `T` is a value type.

20.8.5 The is operator

The `is` operator operates on open types largely following the usual rules (§7.9.9). If either the compile-time type of `e` or `T` is an open type, then a dynamic type check on the run-time types of `e` and `T` is always performed.

20.8.6 The as operator

The `as` operator can be used with a type parameter `T` as the right hand side only if `T` has a class constraint. This restriction is required because the value `null` might be returned as a result of the operator.

```
class X
{
   public T F<T>(object o) where T: Attribute {
      return o as T;          // Ok, T has a class constraint
   }
   public T G<T>(object o) {
      return o as T;          // Error, unconstrained T
   }
}
```

In the current specification for the `as` operator (§7.9.10), for the expression `e as T` the final bullet point states that if no explicit reference conversion is available from the compile-time type of `e` to `T`, a compile-time error occurs. With generics, this rule changes slightly. If either the compile-time type of `e` or `T` is an open type, then no compile-time error occurs in this case; instead a run-time type check occurs.

20.8.7 Exception statements

The usual rules for `throw` (§8.9.5) and `try` (§8.10) statements apply when used with open types:

- The `throw` statement can be used with an expression whose type is given by a type parameter only if that type parameter has `System.Exception` (or a subclass thereof) as a class constraint.
- The type named in a catch clause may be a type parameter only if that type parameter has `System.Exception` (or a subclass thereof) as a class constraint.

20.8.8 The lock statement

The `lock` statement may be used with an expression whose type is given by a type parameter. If the run-time type of the expression is a value type, the locking will have no effect (since the boxed value could not have any other references to it).

20.8.9 The using statement

The `using` statement (§8.13) follows the usual rules: the expression must be implicitly convertible to `System.IDisposable`. If a type parameter is constrained by `System.IDisposable`, then expressions of that type may be used with a `using` statement.

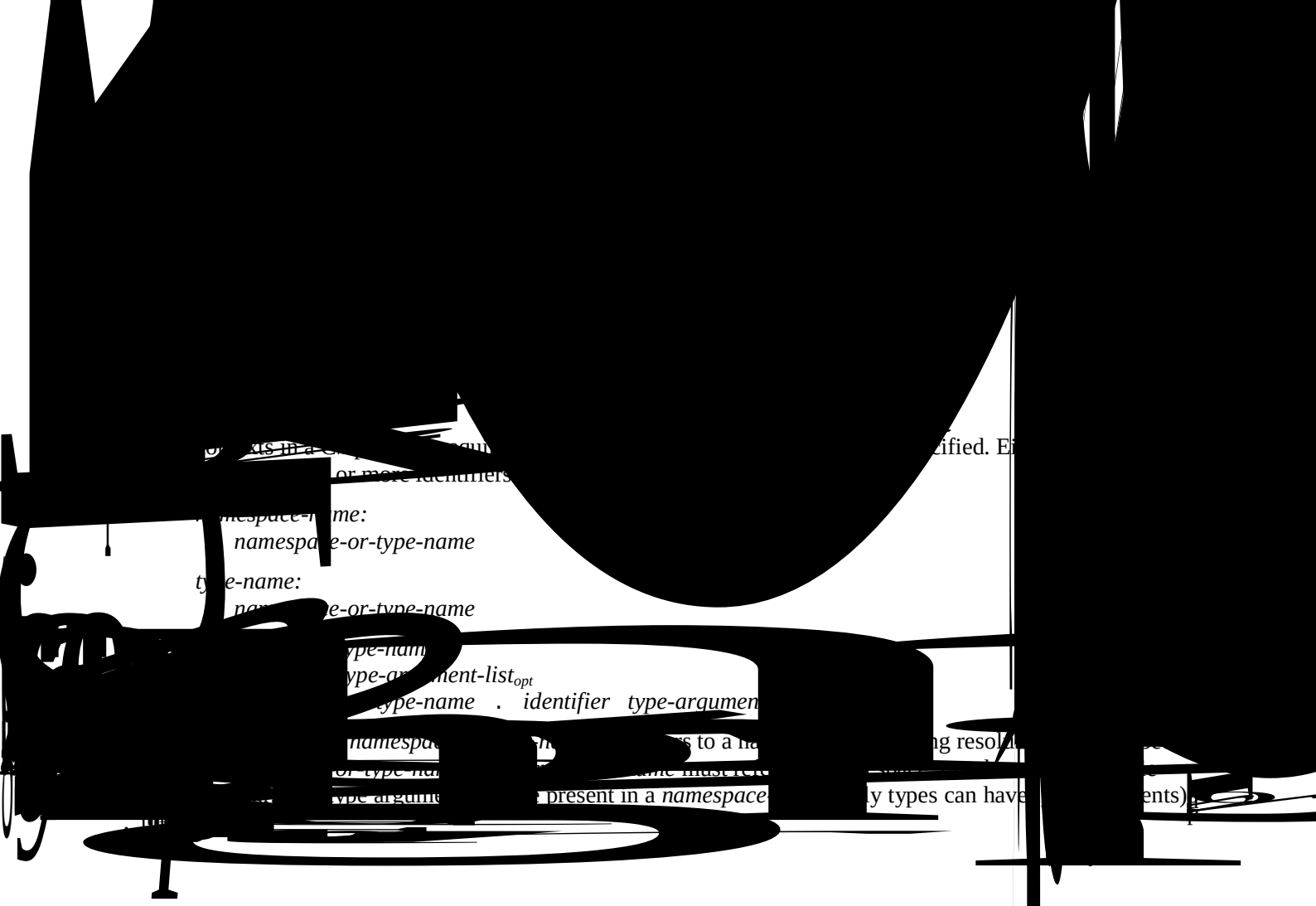
20.8.10 The foreach statement

Given a `foreach` statement of the form:

```
foreach (ElementType element in collection) statement
```

If the `collection` expression is a type that does not implement the collection pattern, but does implement the constructed interface `System.Collections.Generic.IEnumerable<T>` for exactly one type `T`, then the expansion of the `foreach` statement is:

```
IEnumerator<T> enumerator = ((IEnumerable<T>)(collection)).GetEnumerator();
try {
   while (enumerator.MoveNext()) {
      ElementType element = (ElementType)enumerator.Current;
      statement;
   }
}
finally {
   enumerator.Dispose();
}
```

- Otherwise, if `I` is the name of an accessible type in `N` with a matching number of type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
- Otherwise, if the location where the *namespace-or-type-name* occurs is enclosed by a namespace declaration for `N`:
  - If the namespace declaration contains a *using-alias-directive* that associates the name given by `I` with an imported namespace or type, then the *namespace-or-type-name* refers to that namespace or type. If a type argument list was specified, a compile-time error occurs.
  - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type with the name given by `I` and a matching number of type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type with the name given by `I` and a matching number of type parameters, then the *namespace-or-type-name* is ambiguous and an error occurs.
- Otherwise, the *namespace-or-type-name* is undefined and a compile-time error occurs.

- Otherwise, the *namespace-or-type-name* is of the form `N.I` or of the form `N.I<`A_1`, ..., `A_N`>`, where `N` is a *namespace-or-type-name*, `I` is an identifier, and `<`A_1`, ..., `A_N`>` is an optional type argument list. `N` is first resolved as a *namespace-or-type-name*. If the resolution of `N` is not successful, a compile-time error occurs. Otherwise, `N.I` or `N.I<`A_1`, ..., `A_N`>` is resolved as follows:
  - If `N` refers to a namespace and if `I` is the name of a nested namespace in `N`, then the *namespace-or-type-name* refers to that nested namespace. If a type argument list was specified, a compile-time error occurs.
  - Otherwise, if `N` refers to a namespace and `I` is the name of an accessible type in `N` with a matching number of type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, if `N` refers to a (possibly constructed) class or struct type and `I` is the name of an accessible type nested in `N` with a matching number of type parameters, then the *namespace-or-type-name* refers to that type constructed with the given type arguments.
  - Otherwise, `N.I` is an invalid *namespace-or-type-name*, and a compile-time error occurs.

20.9.2 Member lookup

The following replaces §7.3:

A member lookup is the process whereby the meaning of a name in the context of a type is determined. A member lookup may occur as part of evaluating a *simple-name* (§20.9.3) or a *member-access* (§20.9.4) in an expression.

A member lookup of a name `N` in a type `T` is processed as follows:

- First, a set of accessible members named `N` is determined:
  - If `T` is a type parameter, then the set is the union of the sets of accessible members named `N` in each of the types specified as a class constraint or interface constraint for `T`, along with the set of accessible members named `N` in `object`.
  - Otherwise, the set consists of all accessible (§3.5) members named `N` in `T`, including inherited members and the accessible members named `N` in `object`. If `T` is a constructed type, the set of members is obtained by substituting type arguments as described in §20.5.4. Members that include an `override` modifier are excluded from the set.

- Next, members that are hidden by other members are removed from the set. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied:
  - If `M` is a constant, field, property, event, or enumeration member, then all members declared in a base type of `S` are removed from the set.
  - If `M` is a type declaration, then all non-types declared in a base type of `S` are removed from the set, and all type declarations with the same number of type parameters as `M` declared in a base type of `S` are removed from the set.
  - If `M` is a method, then all non-method members declared in a base type of `S` are removed from the set, and all methods with the same signature as `M` declared in a base type of `S` are removed from the set.
- Next, interface members that are hidden by class members are removed from the set. This step only has an effect if `T` is a type parameter and `T` has both a class constraint and one or more interface constraints. For every member `S.M` in the set, where `S` is the type in which the member `M` is declared, the following rules are applied if `S` is a class declaration other than `object`:
  - If `M` is a constant, field, property, event, enumeration member, or type declaration, then all members declared in an interface declaration are removed from the set.
  - If `M` is a method, then all non-method members declared in an interface declaration are removed from the set, and all methods with the same signature as `M` declared in an interface declaration are removed from the set.
- Finally, having removed hidden members, the result of the lookup is determined:
  - If the set consists of a single member that is not a type and not a method, then this member is the result of the lookup.
  - Otherwise, if the set contains only methods, then this group of methods is the result of the lookup.
  - Otherwise, if the set contains only type declarations, then this group of type declarations in the result of the lookup.
  - Otherwise, the lookup is ambiguous, and a compile-time error occurs.

For member lookups in types other than interfaces, and member lookups in interfaces that are strictly single-inheritance (each interface in the inheritance chain has exactly zero or one direct base interface), the effect of the lookup rules is simply that derived members hide base members with the same name or signature. Such single-inheritance lookups are never ambiguous. The ambiguities that can possibly arise from member lookups in multiple-inheritance interfaces are described in §13.2.5.

20.9.3 Simple names

The following replaces §7.5.2:

A *simple-name* consists of an identifier, optionally followed by a type parameter list:

simple-name:
identifier *type-argument-list*$_{opt}$

A *simple-name* of the form `I` or of the form `I<`A_1`, ..., `A_N`>`, where `I` is an identifer and `<`A_1`, ..., `A_N`>` is an optional type argument list, is evaluated and classified as follows:

- If the *simple-name* appears within a *block* and if the *block*'s (or an enclosing block's) local variable declaration space (§3.3) contains a local variable or parameter with the name given by `I`, then the *simple-name* refers to that local variable or parameter and is classified as a variable. If a type argument list was specified, a compile-time error occurs.

- If the *simple-name* appears within the body of a generic method declaration and if that declaration includes a type parameter with the name given by `I`, then the *simple-name* refers to that type parameter. If a type argument list was specified, a compile-time error occurs.
- Otherwise, for each instance type `T` (§20.1.2), starting with the instance type of the immediately enclosing class, struct, or enumeration declaration and continuing with the instance type of each enclosing outer class or struct declaration (if any):
  - If the declaration of `T` includes a type parameter of the name given by `I`, then the *simple-name* refers to that type parameter. If a type argument list was specified, a compile-time error occurs.
  - Otherwise, if a member lookup (§20.9.2) of `I` in `T` produces a match:
    - If `T` is the instance type of the immediately enclosing class or struct type and the lookup identifies one or more methods, the result is a method group with an associated instance expression of `this`. If a type argument list was specified, it is used in calling a generic method (§20.6.3).
    - If `T` is the instance type of the immediately enclosing class or struct type, if the lookup identifies an instance member, and if the reference occurs within the *block* of an instance constructor, an instance method, or an instance accessor, the result is the same as a member access (§20.9.4) of the form `this.I`. If a type argument list was specified, a compile-time error occurs.
    - Otherwise, the result is the same as a member access (§20.9.4) of the form `T.I` or `T.I<`A_1`, ..., `A_N`>`. In this case, it is a compile-time error for the *simple-name* to refer to an instance member.
- Otherwise, for each namespace `N`, starting with the namespace in which the *simple-name* occurs, continuing with each enclosing namespace (if any), and ending with the global namespace, the following steps are evaluated until an entity is located:
  - If `I` is the name of a namespace in `N` and no type argument list was specified, then the *simple-name* refers to that namespace.
  - Otherwise, if `I` is the name of an accessible type in `N` with a matching number of type parameters, then the *simple-name* refers to that type constructed with the given type arguments.
  - Otherwise, if the location where the *simple-name* occurs is enclosed by a namespace declaration for `N`:
    - If the namespace declaration contains a *using-alias-directive* that associates the name given by `I` with an imported namespace or type, then the *simple-name* refers to that namespace or type. If a type argument list was specified, a compile-time error occurs.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain exactly one type with the name given by `I` and a matching number of type parameters, then the *simple-name* refers to that type constructed with the given type arguments.
    - Otherwise, if the namespaces imported by the *using-namespace-directive*s of the namespace declaration contain more than one type with the name given by `I` and a matching number of type parameters, then the *simple-name* is ambiguous and an error occurs.
- Otherwise, the name given by the *simple-name* is undefined and a compile-time error occurs.

20.9.4 Member access

The following replaces §7.5.4:

A *member-access* consists of a *primary-expression* or a *predefined-type*, followed by a "`.`" token, followed by an *identifier*, optionally followed by a *type-argument-list*.

member-access:
primary-expression `.` *identifier type-argument-list*$_{opt}$
predefined-type `.` *identifier type-argument-list*$_{opt}$

predefined-type: one of

```
bool     byte     char     decimal  double   float    int      long
object   sbyte    short    string   uint     ulong    ushort
```

A *member-access* of the form `E.I` or of the form `E.I<`A_1`, ..., `A_N`>`, where `E` is a *primary-expression* or a *predefined-type*, `I` is an *identifier*, and `<`A_1`, ..., `A_N`>` is an optional *type-argument-list*, is evaluated and classified as follows:

- If `E` is a namespace and `I` is the name of a nested namespace in `E`, then the result is that namespace. If a type argument list was specified, a compile-time error occurs.
- If `E` is a namespace and `I` is the name of an accessible type in `E`, then the result is that type constructed with the given type arguments. If the number of type arguments does not match the number of type parameters, a compile-time error occurs.
- If `E` is a *predefined-type* or a *primary-expression* classified as a type, if `E` is not a type parametef

- First, if E is a property or indexer access, then the value of the property or indexer access is obtained (§7.1.1) and E is reclassified as a value.
- If I identifies one or more methods, then the result is a method group with an associated instance expression of E. If a type argument list was specified, it is used in calling a generic method (§20.6.3).
- If I identifies an instance property, an instance field, or an instance event, and if a type argument list was specified, a compile-time error occurs.
- If I identifies an instance property, then the result is a property access with an associated instance expression of E.
- If T is a *class-type* and I identifies an instance field of that *class-type*:
  - If the value of E is `null`, then a `System.NullReferenceException` is thrown.
  - Otherwise, if the field is `readonly` and the reference occurs outside an instance constructor of the class in which the field is declared, then the result is a value, namely the value of the field I in the object referenced by E.
  - Otherwise, the result is a variable, namely the field I in the object referenced by E.
- If T is a *struct-type* and I identifies an instance field of that *struct-type*:
  - If E is a value, or if the field is `readonly` and the reference occurs outside an instance constructor of the struct in which the field is declared, then the result is a value, namely the value of the field I in the struct instance given by E.
  - Otherwise, the result is a variable, namely the field I in the struct instance given by E.
- If I identifies an instance event:
  - If the reference occurs within the class or struct in which the event is declared, and the event was declared without *event-accessor-declarations* (§10.7), then E.I is processed exactly as if I was an instance field.
  - Otherwise, the result is an event access with an associated instance expression of E.

- Otherwise, E.I is an invalid member reference, and a compile-time error occurs.

20.9.5 Method invocations

The following replaces the part of §7.5.5.1 that describes compile-time processing of a method invocation:

The compile-time processing of a method invocation of the form M(A), where M is a method group (possibly including a *type-argument-list*), and A is an optional *argument-list*, consists of the following steps:

- The set of candidate methods for the method invocation is constructed. For each method F associated with the method group M:
  - If F is non-generic, F is a candidate when:
    - M has no type argument list, and
    - F is applicable with respect to A (§7.4.2.1).
  - If F is generic and M has no type argument list, F is a candidate when:
    - Type inference (§20.6.4) succeeds, inferring a list of type arguments for the call, and
    - Once the inferred type arguments are substituted for the corresponding method type parameters, the parameter list of F is applicable with respect to A (§7.4.2.1), and

- The parameter list of `F`, after substituting type arguments, is not the same as an applicable non-generic method, possibly in expanded form (§7.4.2.1), declared in the same type as `F`.

o If `F` is generic and `M` includes a type argument list, `F` is a candidate when:

- `F` has the same number of method type parameters as were supplied in the type argument list, and
- Once the type arguments are substituted for the corresponding method type parameters, the parameter list of `F` is applicable with respect to `A` (§7.4.2.1).

- The set of candidate methods is reduced to contain only methods from the most derived types: For each method `C.F` in the set, where `C` is the type in which the method `F` is declared, all methods declared in a base type of `C` are removed from the set.
- If the resulting set of candidate methods is empty, then no applicable methods exist, and a compile-time error occurs. If the candidate methods are not all declared in the same type, the method invocation is ambiguous, and a compile-time error occurs (this latter situation can only occur for an invocation of a method in an interface that has multiple direct base interfaces, as described in §13.2.5).
- The best method of the set of candidate methods is identified using the overload resolution rules of §7.4.2. If a single best method cannot be identified, the method invocation is ambiguous, and a compile-time error occurs. When performing overload resolution, the parameters of a generic method are considered after substituting the type arguments (supplied or inferred) for the corresponding method type parameters.

s • Finalevalidation of the chosen best met r

f n t o§ h c s s

nt tn g o dnit c e n s. es o a e g r

- The overload resolution step is not performed. Instead, the set of candidates must include exactly one method that is compatible (§15.1) with `D` (following substitution of type parameters with type arguments), and this method becomes the one to which the newly created delegate refers. If no matching method exists, or if more than one matching method exists, a compile-time error occurs.

  - If the selected method is an instance method, the instance expression associated with `E` determines the target object of the delegate.
  - The result is a value of type `D`, namely a newly created delegate that refers to the selected method and target object.

- Otherwise, if `E` is a value of a *delegate-type*:
  - `D` and `E` must be compatible (§15.1); otherwise, a compile-time error occurs.
  - The result is a value of type `D`, namely a newly created delegate that refers to the same invocation list as `E`.
- Otherwise, the delegate creation expression is invalid, and a compile-time error occurs.

20.10 Right-shift grammar changes

The syntax for generics uses the < and > characters to delimit type parameters and type arguments (similar to the syntax used in C++ for templates). Constructed types sometimes nest, as in `List<Nullable<int>>`, but there is a subtle grammatical problem with such constructs: the lexical grammar will combine the last two characters of this construct into the single token >> (the right shift operator), rather than producing two > tokens, which the syntactic grammar would require. Although a possible solution is to put a space in between the two > characters, this is awkward and confusing, and does not add to the clarity of the program in any way.

In order to allow these natural constructs, and to maintain a simple lexical grammar, the >> and >= tokens are removed from the lexical grammar and replaced with *right-shift* and *right-shift-assignment* productions:

operator-or-punctuator: one of

```
{      }      [      ]      (      )      .      ,      :      ;
+      -      *      /      %      &      |      ^      !      ~
=      <      >      ?      ++     --     &&     ||     ==     ->
!=     <=     >=     +=     -=     *=     /=     %=     &=     |=
^=     <<     <<=
```

right-shift:

```
> >
```

right-shift-assignment:

```
> >=
```

Unlike other productions in the syntactic grammar, no characters of any kind (not even whitespace) are allowed between the tokens in the *right-shift* and *right-shift-assignment* productions.

The following productions are modified to use *right-shift* and *right-shift-assignment*:

shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression *right-shift* *additive-expression*

assignment-operator:
=
+=
-=
*=
/=
%=
&=
|=
^=
<<=
right-shift-assignment

overloadable-binary-operator:
+
-
*
/
%
&
|
^
<<
right-shift
==
!=
>
<
>=
<=

Chapter

21. Anonymous methods

21.1 Anonymous method expressions

An *anonymous-method-expression* defines an ***anonymous method*** and evaluates to a special value referencing the method:

primary-no-array-creation-expression:
…
anonymous-method-expression

anonymous-method-expression:
`delegate` *anonymous-method-signature*$_{opt}$ *block*

anonymous-method-signature:
`(` *anonymous-method-parameter-list*$_{opt}$ `)`

anonymous-method-parameter-list:
anonymous-method-parameter
anonymous-method-parameter-list `,` *anonymous-method-parameter*

anonymous-method-parameter:
parameter-modifier$_{opt}$ *type* *identifier*

An *anonymous-method-expression* is classified as a value with special conversion rules (§21.3). The value does not have a type but can be implicitly converted to a compatible delegate type.

The *anonymous-method-expression* defines a new declaration space for parameters, locals and constants and a new declaration space for labels (§3.3).

21.2 Anonymous method signatures

The optional *anonymous-method-signature* defines the names and types of the formal parameters for the anonymous method. The scope of the parameters of the anonymous method is the *block*. It is a compile-time error for the name of a parameter of the anonymous method to match the name of a local variable, local constant or parameter whose scope includes the *anonymous-method-expression*.

If an *anonymous-method-expression* has an *anonymous-method-signature*, then the set of compatible delegate types is restricted to those that have the same parameter types and modifiers in the same order (§21.3). If an *anonymous-method-expression* doesn't have an *anonymous-method-signature*, then the set of compatible delegate types is restricted to those that have no `out` parameters.

Note that an *anonymous-method-signature* cannot include attributes or a parameter array. Nevertheless, an *anonymous-method-signature* may be compatible with a delegate type whose parameter list contains a parameter array.

21.3 Anonymous method conversions

An *anonymous-method-expression* is classified as a value with no type. An *anonymous-method-expression* may be used in a *delegate-creation-expression* (§21.3.1). All other valid uses of an *anonymous-method-expression* depend on the implicit conversions defined here.

An implicit conversion (§6.1) exists from an *anonymous-method-expression* to any ***compatible*** delegate type. If `D` is a delegate type, and `A` is an *anonymous-method-expression*, then `D` is compatible with `A` if and only if the following two conditions are met.

- First, the parameter types of `D` must be compatible with `A`:
  - If `A` does not contain an *anonymous-method-signature*, then `D` may have zero or more parameters of any type, as long as no parameter of `D` has the `out` parameter modifier.
  - If `A` has an *anonymous-method-signature*, then `D` must have the same number of parameters, each parameter of `A` must be of the same type as the corresponding parameter of `D`, and the presence or absence of the `ref` or `out` modifier on each parameter of `A` must match the corresponding parameter of `D`. Whether the last parameter of `D` is a *parameter-array* is not relevant to the compatibility of `A` and `D`.
- Second, the return type of `D` must be compatible with `A`. For these rules, `A` is not considered to contain the *block* of any other anonymous methods.
  - If `D` is declared with a `void` return type, then any `return` statement contained in `A` may not specify an expression.
  - If `D` is declared with a return type of `R`, then any `return` statement contained in `A` must specify an expression which is implicitly convertible (§6.1) to `R`. Furthermore, the end-point of the *block* of `A` must not be reachable.

Besides the implicit conversions to compatible delegate types, no other conversions exist from an *anonymous-method-expression*, even to the type `object`.

The following examples illustrate these rules:

```
delegate void D(int x);

D d1 = delegate { };                  // Ok
D d2 = delegate() { };                // Error, signature mismatch
D d3 = delegate(long x) { };          // Error, signature mismatch
D d4 = delegate(int x) { };           // Ok
D d5 = delegate(int x) { return; };   // Ok
D d6 = delegate(int x) { return x; }; // Error, return type mismatch

delegate void E(out int x);

E e1 = delegate { };                  // Error, E has an out parameter
E e2 = delegate(out int x) { x = 1; }; // Ok
E e3 = delegate(ref int x) { x = 1; }; // Error, signature mismatch

delegate int P(params int[] a);

P p1 = delegate { };                  // Error, end of block reachable
P p2 = delegate { return; };          // Error, return type mismatch
P p3 = delegate { return 1; };        // Ok
P p4 = delegate { return "Hello"; };  // Error, return type mismatch
P p5 = delegate(int[] a) {            // Ok
   return a[0];
};
P p6 = delegate(params int[] a) {     // Error, params modifier
   return a[0];
};
P p7 = delegate(int[] a) {            // Error, return type mismatch
   if (a.Length > 0) return a[0];
   return "Hello";
};

delegate object Q(params int[] a);
```

```
Q q1 = delegate(int[] a) {              // Ok
   if (a.Length > 0) return a[0];
   return "Hello";
};
```

21.3.1 Delegate creation expression

A *delegate-creation-expression* (§7.5.10.3) can be used as an alternate syntax for converting an anonymous method to a delegate type. If the *expression* used as the argument of a *delegate-creation-expression* is an *anonymous-method-expression*, then the anonymous method is converted to the given delegate type using the implicit conversion rules defined above. For example, if `D` is a delegate type, then the expression

```
new D(delegate { Console.WriteLine("hello"); })
```

is equivalent to the expression

```
(D) delegate { Console.WriteLine("hello"); }
```

21.4 Anonymous method blocks

The *block* of an *anonymous-method-expression* is subject to the following rules:

- If the anonymous method includes a signature, the parameters specified in the signature are available in the *block*. If the anonymous method has no signature it can be converted to a delegate type having parameters (§21.3), but the parameters cannot be accessed in the *block*.
- Except for `ref` or `out` parameters specified in the signature (if any) of the nearest enclosing anonymous method, it is a compile-time error for the *block* to access a `ref` or `out` parameter.
- When the type of `this` is a struct type, it is a compile-time error for the *block* to access `this`. This is true whether the access is explicit (as in `this.x`) or implicit (as in `x` where `x` is an instance member of the struct). This rule simply prohibits such access and does not affect whether member lookup results in a member of the struct.
- The *block* has access to the outer variables (§21.5) of the anonymous method. Access of an outer variable will reference the instance of the variable that is active at the time the *anonymous-method-expression* is evaluated (§21.6).
- It is a compile-time error for the *block* to contain a `goto` statement, `break` statement, or `continue` statement whose target is outside the *block* or within the *block* of a contained anonymous method.
- A `return` statement in the *block* returns control from an invocation of the nearest enclosing anonymous method, not from the enclosing function member. An expression specified in a `return` statement must be compatible with the delegate type to which the nearest enclosing *anonymous-method-expression* is converted (§21.3).

It is explicitly unspecified whether there is any way to execute the *block* of an anonymous method other than through evaluation and invocation of the *anonymous-method-expression*. In particular, a compiler may choose to implement an anonymous method by synthesizing one or more named methods or types. The names of any such synthesized elements must be in the space reserved for compiler use: the names must contain two consecutive underscore characters.

21.5 Outer variables

Any local variable, value parameter, or parameter array whose scope includes the *anonymous-method-expression* is called an ***outer variable*** of the *anonymous-method-expression*. In an instance function member of a class, the `this` value is considered a value parameter and is an outer variable of any *anonymous-method-expression* contained within the function member.

21.5.1 Captured outer variables

When an outer variable is referenced by an anonymous method, the outer variable 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 (§5.1.7). 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.

In the example

```
using System;

delegate int D();

class Test
{
   static D F() {
      int x = 0;
      D result = delegate { return ++x; }
      return result;
   }

   static void Main() {
      D d = F();
      Console.WriteLine(d());
      Console.WriteLine(d());
      Console.WriteLine(d());
   }
}
```

the local variable `x` is captured by the anonymous method, and the lifetime of `x` is extended at least until the delegate returned from `F` becomes eligible for garbage collection (which doesn't happen until the very end of the program). Since each invocation of the anonymous method operates on the same instance of `x`, the output of the example is:

```
1
2
3
```

When a local variable or a value parameter is captured by an anonymous method, the local variable or parameter is no longer considered to be a fixed variable (§18.3), but is instead considered to be a moveable variable. Thus any `unsafe` code that takes the address of a captured outer variable must first use the `fixed` statement to fix the variable.

21.5.2 Instantiation of local variables

A local variable is considered to be ***instantiated*** when execution enters the scope of the variable. For example, when the following method is invoked, the local variable `x` is instantiated and initialized three times—once for each iteration of the loop.

```
static void F() {
   for (int i = 0; i < 3; i++) {
      int x = i * 2 + 1;
      ...
   }
}
```

However, moving the declaration of `x` outside the loop results in a single instantiation of `x`:

```
static void F() {
   int x;
   for (int i = 0; i < 3; i++) {
      x = i * 2 + 1;
      ...
   }
}
```

Ordinarily, there is no way to observe exactly how often a local variable is instantiated—because the lifetimes of the instantiations are disjoint, it is possible for each instantation to simply use the same storage location. However, when an anonymous method captures a local variable, the effects of instantiation become apparent. The example

```
using System;

delegate void D();

class Test
{
   static D[] F() {
      D[] result = new D[3];
      for (int i = 0; i < 3; i++) {
         int x = i * 2 + 1;
         result[i] = delegate { Console.WriteLine(x); };
      }
      return result;
   }

   static void Main() {
      foreach (D d in F()) d();
   }
}
```

produces the output:

```
1
3
5
```

However, when the declaration of `x` is moved outside the loop:

```
static D[] F() {
   D[] result = new D[3];
   int x;
   for (int i = 0; i < 3; i++) {
      x = i * 2 + 1;
      result[i] = delegate { Console.WriteLine(x); };
   }
   return result;
}
```

the output is:

```
5
5
5
```

Note that the three delegates created in the version of `F` directly above will be equal according to the equality operator (§21.7). Furthermore, note that the compiler is permitted (but not required) to optimize the three instantiations into a single delegate instance (§21.6).

It is possible for anonymous method delegates to share some captured variables yet have separate instances of others. For example, if `F` is changed to

```
static D[] F() {
   D[] result = new D[3];
   int x = 0;
   for (int i = 0; i < 3; i++) {
      int y = 0;
      result[i] = delegate { Console.WriteLine("{0} {1}", ++x, ++y); };
   }
   return result;
}
```

the three delegates capture the same instance of `x` but separate instances of `y`, and the output is:

```
1 1
2 1
3 1
```

Separate anonymous methods can capture the same instance of an outer variable. In the example:

```
using System;

delegate void Setter(int value);

delegate int Getter();

class Test
{
   static void Main() {
      int x = 0;
      Setter s = delegate(int value) { x = value; };
      Getter g = delegate { return x; };
      s(5);
      Console.WriteLine(g());
      s(10);
      Console.WriteLine(g());
   }
}
```

the two anonymous methods capture the same instance of the local variable `x`, and they can thus "communicate" through that variable. The output of the example is:

```
5
10
```

21.6 Anonymous method evaluation

The run-time evaluation of an *anonymous-method-expression* produces a delegate instance which references the anonymous method and the (possibly empty) set of captured outer variables that are active at the time of the evaluation. When a delegate resulting from an *anonymous-method-expression* is invoked, the body of the anonymous method is executed. The code in the body is executed using the set of captured outer variables referenced by the delegate.

The invocation list of a delegate produced from an *anonymous-method-expression* contains a single entry. The exact target object and target method of the delegate are unspecified. In particular, it is unspecified whether the target object of the delegate is `null`, the `this` value of the enclosing function member, or some other object.

Evaluation of sematically identical *anonymous-method-expression*s with the same (possibly empty) set of captured outer variable instances is permitted (but not required) to return the same delegate instance. The term sematically identical is used here to mean that execution of the anonymous methods will, in all cases, produce the same effects given the same arguments. This rule permits code such as the following to be optimized.

```
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 void F(double[] a, double[] b) {
      a = Apply(a, delegate(double x) { return Math.Sin(x); });
      b = Apply(b, delegate(double y) { return Math.Sin(y); });
      ...
   }
}
```

Since the two anonymous method delegates have the same (empty) set of captured outer variables, and since the anonymous methods are semantically identical, the compiler is permitted to have the delegates refer to the same target method. Indeed, the compiler is permitted to return the very same delegate instance from both anonymous method expressions.

21.7 Delegate instance equality

The following rules govern the results produced by the equality operators (§7.9.8) and the `Object.Equals` method for anonymous method delegate instances:

- Delegate instances produced from evaluation of semantically identical *anonymous-method-expression*s with the same (possibly empty) set of captured outer variable instances are permitted (but not required) to be equal.
- Delegate instances produced from evaluation of semantically different *anonymous-method-expression*s or having different sets of captured outer variable instances are never equal.

21.8 Definite assignment

The definite assignment state of a parameter of an anonymous method is the same as for a parameter of a named method. That is, reference parameters and value parameters are initially definitely assigned and output parameters are initially unassigned. Furthermore, output parameters must be definitely assigned before the anonymous method returns normally (§5.1.6).

The definite assignment state of an outer variable *v* on the control transfer to the *block* of an *anonymous-method-expression* is the same as the definite assignment state of *v* before the *anonymous-method-expression*. That is, definite assignment of outer variables is inherited from the context of the *anonymous-method-expression*. Within the *block* of an *anonymous-method-expression*, definite assignment evolves as in a normal block (§5.3.3).

The definite assignment state of a variable *v* after an *anonymous-method-expression* is the same as its definite assignment state before the *anonymous-method-expression*.

The example

```
delegate bool Filter(int i);

void F() {
   int max;

   // Error, max is not definitely assigned
   Filter f = delegate(int n) { return n < max; }

   max = 5;
   DoWork(f);
}
```


generates a compile-time error since `max` is not definitely assigned where the anonymous method is declared. The example

```
delegate void D();

void F() {
   int n;
   D d = delegate { n = 1; };

   d();

   // Error, n is not definitely assigned
   Console.WriteLine(n);
}
```

also generates a compile-time error since the assignment to `n` in the anonymous method has no affect on the definite assignment state of `n` outside the anonymous method.

21.9 Method group conversions

Similar to the implicit anonymous method conversions described in §21.3, an implicit conversion exists from a method group (§7.1) to a compatible delegate type.

Given a method group `E` and a delegate type `D`, if a delegate creation expression (§7.5.10.3 and §20.9.6) of the form `new D(E)` is permitted, then an implicit conversion from `E` to `D` also exists, and the result of that conversion is exactly equivalent to writing `new D(E)`.

In the example

```
using System;
using System.Windows.Forms;

class AlertDialog
{
   Label message = new Label();
   Button okButton = new Button();
   Button cancelButton = new Button();`

   public AlertDialog() {
      okButton.Click += new EventHandler(OkClick);
      cancelButton.Click += new EventHandler(CancelClick);
      ...
   }

   void OkClick(object sender, EventArgs e) {
      ...
   }

   void CancelClick(object sender, EventArgs e) {
      ...
   }
}
```

the constructor creates two delegate instances using the `new` operator. Implicit method group conversions permit this to be shortened to

```
public AlertDialog() {
   okButton.Click += OkClick;
   cancelButton.Click += CancelClick;
   ...
}
```

As with all other implicit and explicit conversions, the cast operator can be used to explicitly perform a particular conversion. Thus, the example

```
object obj = new EventHandler(myDialog.OkClick);
```


could instead be written

```
object obj = (EventHandler)myDialog.OkClick;
```

Method groups and anonymous method expressions may influence overload resolution, but do not participate in type inferencing. See §20.6.4 for further details.

21.10 Implementation example

This section describes a possible implementation of anonymous methods in terms of standard C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation, nor is it the only one possible.

The remainder of this section gives several examples of code that contains anonymous methods with different characteristics. For each example, a corresponding translation to code that uses only standard C# constructs is provided. In the examples, the identifier `D` is assumed by represent the following delegate type:

```
public delegate void D();
```

The simplest form of an anonymous method is one that captures no outer variables:

```
class Test
{
   static void F() {
      D d = delegate { Console.WriteLine("test"); };
   }
}
```

This can be translated to a delegate instantiation that references a compiler generated static method in which the code of the anonymous method is placed:

```
class Test
{
   static void F() {
      D d = new D(__Method1);
   }

   static void __Method1() {
      Console.WriteLine("test");
   }
}
```

In the following example, the anonymous method references instance members of `this`:

```
class Test
{
   int x;

   void F() {
      D d = delegate { Console.WriteLine(x); };
   }
}
```

This can be translated to a compiler generated instance method containing the code of the anonymous method:

```
class Test
{
   int x;

   void F() {
      D d = new D(__Method1);
   }

   void __Method1() {
      Console.WriteLine(x);
   }
}
```

In this example, the anonymous method captures a local variable:

```
class Test
{
   void F() {
      int y = 123;
      D d = delegate { Console.WriteLine(y); };
   }
}
```

The lifetime of the local variable must now be extended to at least the lifetime of the anonymous method delegate. This can be achieved by "lifting" the local variable into a field of a compiler generated class. Instantiation of the local variable (§21.5.2) then corresponds to creating an instance of the compiler generated class, and accessing the local variable corresponds to accessing a field in the instance of the compiler generated class. Furthermore, the anonymous method becomes an instance method of the compiler generated class:

```
class Test
{
   void F() {
      __locals1 = new __Locals1();
      __locals1.y = 123;
      D d = new D(__locals1.__Method1);
   }

   class __Locals1
   {
      public int y;

      public void __Method1() {
         Console.WriteLine(y);
      }
   }
}
```

Finally, the following anonymous method captures `this` as well as two local variables with different lifetimes:

```
class Test
{
   int x;

   void F() {
      int y = 123;
      for (int i = 0; i < 10; i++) {
         int z = i * 2;
         D d = delegate { Console.WriteLine(x + y + z); };
      }
   }
}
```

Here, a compiler generated class is created for each statement block in which locals are captured such that the locals in the different blocks can have independent lifetimes. An instance of `__Locals2`, the compiler generated class for the inner statement block, contains the local variable `z` and a field that references an instance of `__Locals1`. An instance of `__Locals1`, the compiler generated class for the outer statement block, contains the local variable `y` and a field that references `this` of the enclosing function member. With these data structures it is possible to reach all captured outer variables through an instance of `__Local2`, and the code of the anonymous method can thus be implemented as an instance method of that class.

```
class Test
{
   void F() {
      __locals1 = new __Locals1();
      __locals1.__this = this;
      __locals1.y = 123;
      for (int i = 0; i < 10; i++) {
         __locals2 = new __Locals2();
         __locals2.__locals1 = __locals1;
         __locals2.z = i * 2;
         D d = new D(__locals2.__Method1);
      }
   }

   class __Locals1
   {
      public Test __this;
      public int y;
   }

   class __Locals2
   {
      public __Locals1 __locals1;
      public int z;

      public void __Method1() {
         Console.WriteLine(__locals1.__this.x + __locals1.y + z);
      }
   }
}
```

Chapter

22. Iterators

22.1 Iterator blocks

An iterator block is a *block* (§8.2) that yields an ordered sequence of values. An iterator block 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 block may be used as a *method-body*, *operator-body* or *accessor-body* as long as the return type of the corresponding function member is one of the enumerator interfaces (§22.1.1) or one of the enumerable interfaces (§22.1.2).

Iterator blocks are not a distinct element in the C# grammar. They are restricted in several ways and have a major effect on the semantics of a function member declaration, but they are grammatically just blocks.

When a function member is implemented using an iterator block, it is a compile-time error for the formal parameter list of the function member to specify any `ref` or `out` parameters.

It is a compile-time error for a `return` statement to appear in an iterator block (but `yield return` statements are permitted).

It is a compile-time error for an iterator block to contain an unsafe context (§18.1). An iterator block always defines a safe context, even when its declaration is nested in an unsafe context.

22.1.1 Enumerator interfaces

The ***enumerator interfaces*** are the non-generic interface `System.Collections.IEnumerator` and all instantiations of the generic interface `System.Collections.Generic.IEnumerator<T>`. In this chapter, these interfaces are referenced as `IEnumerator` and `IEnumerator<T>`, respectively.

22.1.2 Enumerable interfaces

The ***enumerable interfaces*** are the non-generic interface `System.Collections.IEnumerable` and all instantiations of the generic interface `System.Collections.Generic.IEnumerable<T>`. In this chapter, these interfaces are referenced as `IEnumerable` and `IEnumerable<T>`, respectively.

22.1.3 Yield type

An iterator block produces a sequence of values, all of the sam

22.1.4 This access

Within an iterator block of an instance member of a class, the expression `this` is classified as a value. The type of the value is the class within which the usage occurs, and the value is a reference to the object for which the member was invoked.

Within an iterator block of an instance member of a struct, the expression `this` is classified as a variable. The type of the variable is the struct within which the usage occurs. The variable represents a *copy* of the struct for which the member was invoked. The `this` variable in an iterator block of an instance member of a struct behaves exactly the same as a *value* parameter of the struct type.

22.2 Enumerator objects

When a function member returning an enumerator interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an ***enumerator object*** is created and returned. This object encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerator object has the following characteristics:

- It implements `IEnumerator` and `IEnumerator<T>`, where `T` is the yield type of the iterator block.
- It implements `System.IDisposable`.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.
- It has four potential states, ***before***, ***running***, ***suspended***, and ***after***, and is initially in the ***before*** state.

An enumerator object is typically an instance of a compiler-generated enumerator class that encapsulates the code in the iterator block and implements the enumerator interfaces, but other methods of implementation are possible. If an enumerator class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (§2.4.2).

An enumerator object may implement more interfaces than those specified above.

The following sections describe the exact behavior of the `MoveNext`, `Current`, and `Dispose` members of the `IEnumerable` and `IEnumerable<T>` interface implementations provided by an enumerator object.

Note that enumerator objects do not support the `IEnumerator.Reset` method. Invoking this method will throw a `System.NotSupportedException`.

22.2.1 The MoveNext method

The `MoveNext` method of an enumerator object encapsulates the code of an iterator block. Invoking the `MoveNext` method executes code in the iterator block and sets the `Current` property of the enumerator object as appropriate. The precise action performed by `MoveNext` depends on the state of the enumerator object when `MoveNext` is invoked:

- If the state of the enumerator object is ***before***, invoking `MoveNext`:
  - Changes the state to ***running***.
  - Initializes the parameters (including `this`) of the iterator block to the argument values and instance value saved when the enumerator object was intialized.
  - Executes the iterator block from the beginning until execution is interrupted (as described below).
- If the state of the enumerator object is ***running***, the result of invoking `MoveNext` is unspecified.
- If the state of the enumerator object is ***suspended***, invoking `MoveNext`:

  - Changes the state to ***running***.
  - Restores the values of all local variables and parameters (including `this`) to the values saved when execution of the iterator block was last suspended. Note that the contents of any objects referenced by these variables may have changed since the previous call to `MoveNext`.
  - Resumes execution of the iterator block immediately following the `yield return` statement that caused the suspension of execution and continues until execution is interrupted (as described below).

- If the state of the enumerator object is ***after***, invoking `MoveNext` returns `false`.

When `MoveNext` executes the iterator block, execution can be interrupted in four ways: By a `yield return` statement, by a `yield break` statement, by encountering the end of the iterator block, and by an exception being thrown and propagated out of the iterator block.

- When a `yield return` statement is encountered (§22.4):
  - The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
  - Execution of the iterator body is suspended. The values of all local variables and parameters (including `this`) are saved, as is the location of this `yield return` statement. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are *not* executed at this time.
  - The state of the enumerator object is changed to ***suspended***.
  - The `MoveNext` method returns `true` to its caller, indicating that the iteration successfully advanced to the next value.
- When a `yield break` statement is encountered (§22.4):
  - If the `yield break` statement is within one or more `try` blocks, the associated `finally` blocks are executed.
  - The state of the enumerator object is changed to ***after***.
  - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When the end of the iterator body is encountered:
  - The state of the enumerator object is changed to ***after***.
  - The `MoveNext` method returns `false` to its caller, indicating that the iteration is complete.
- When an exception is thrown and propagated out of the iterator block:
  - Appropriate `finally` blocks in the iterator body will have been executed by the exception propagation.
  - The state of the enumerator object is changed to ***after***.
  - The exception propagation continues to the caller of the `MoveNext` method.

22.2.2 The Current property

An enumerator object's `Current` property is affected by `yield return` statements in the iterator block.

When an enumerator object is in the ***suspended*** state, the value of `Current` is the value set by the last call to `MoveNext`. When an enumerator object is in the ***before***, ***running***, or ***after*** states, the result of accessing `Current` is unspecified.

For an iterator block with a yield type other than `object`, the result of accessing `Current` through the enumerator object's `IEnumerable` implementation corresponds to accessing `Current` through the enumerator object's `IEnumerator<T>` implementation and casting the result to `object`.

22.2.3 The Dispose method

The `Dispose` method is used to clean up the iteration by bringing the enumerator object to the ***after*** state.

- If the state of the enumerator object is ***before***, invoking `Dispose` changes the state to ***after***.
- If the state of the enumerator object is ***running***, the result of invoking `Dispose` is unspecified.
- If the state of the enumerator object is ***suspended***, invoking `Dispose`:
  - Changes the state to ***running***.
  - Executes any finally blocks as if the last executed `yield return` statement were a `yield break` statement. If this causes an exception to be thrown and propagated out of the iterator body, the state of the enumerator object is set to ***after*** and the exception is propagated to the caller of the `Dispose` method.
  - Changes the state to ***after***.
- If the state of the enumerator object is ***after***, invoking `Dispose` has no affect.

22.3 Enumerable objects

When a function member returning an enumerable interface type is implemented using an iterator block, invoking the function member does not immediately execute the code in the iterator block. Instead, an ***enumerable object*** is created and returned. The enumerable object's `GetEnumerator` method returns an enumerator object that encapsulates the code specified in the iterator block, and execution of the code in the iterator block occurs when the enumerator object's `MoveNext` method is invoked. An enumerable object has the following characteristics:

- It implements `IEnumerable` and `IEnumerable<T>`, where `T` is the yield type of the iterator block.
- It is initialized with a copy of the argument values (if any) and instance value passed to the function member.

An enumerable object is typically an instance of a compiler-generated enumerable class that encapsulates the code in the iterator block and implements the enumerable interfaces, but other methods of implementation are possible. If an enumerable class is generated by the compiler, that class will be nested, directly or indirectly, in the class containing the function member, it will have private accessibility, and it will have a name reserved for compiler use (§2.4.2).

An enumerable object may implement more interfaces than those specified above. In particular, an enumerable object may also implement `IEnumerator` and `IEnumerator<T>`, enabling it to serve as both an enumerable and an enumerator. In that type of implementation, the first time an enumerable object's `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another.

22.3.1 The GetEnumerator method

An enumerable object provides an implementation of the `GetEnumerator` methods of the `IEnumerable` and `IEnumerable<T>` interfaces. The two `GetEnumerator` methods share a common implementation that aquires and returns an available enumerator object. The enumerator object is initialized with the argument values and instance value saved when the enumerable object was initialized, but otherwise the enumerator object functions as described in §22.2.

22.4 The yield statement

The `yield` statement is used in an iterator block to yield a value to the enumerator object or to signal the end of the iteration.

embedded-statement:
...
yield-statement

yield-statement:
`yield return` *expression* `;`
`yield break ;`

To ensure compatibility with existing programs, `yield` is not a reserved word, and `yield` has special meaning only when it is used immediately before a `return` or `break` keyword. In other contexts, `yield` can be used as an identifier.

The are several restrictions on where a `yield` statement can appear, as described in the following.

- It is a compile-time error for a `yield` statement (of either form) to appear outside a *method-body*, *operator-body* or *accessor-body*
- It is a compile-time error for a `yield` statement (of either form) to appear inside an anonymous method.
- It is a compile-time error for a `yield` statement (of either form) to appear in the `finally` clause of a `try` statement.
- It is a compile-time error for a `yield return` statement to appear anywhere in a `try` statement that contains `catch` clauses.

The following example shows some valid and invalid uses of `yield` statements.

```
delegate IEnumerable<int> D();

IEnumerator<int> GetEnumerator() {
   try {
      yield return 1;      // Ok
      yield break;         // Ok
   }
   finally {
      yield return 2;      // Error, yield in finally
      yield break;         // Error, yield in finally
   }

   try {
      yield return 3;      // Error, yield return in try...catch
      yield break;         // Ok
   }
   catch {
      yield return 4;      // Error, yield return in try...catch
      yield break;         // Ok
   }

   D d = delegate {
      yield return 5;      // Error, yield in an anonymous method
   };
}

int MyMethod() {
   yield return 1;         // Error, wrong return type for an iterator block
}
```

An implicit conversion (§6.1) must exist from the type of the expression in the `yield return` statement to the yield type (§22.1.3) of the iterator block.

A `yield return` statement is executed as follows:

- The expression given in the statement is evaluated, implicitly converted to the yield type, and assigned to the `Current` property of the enumerator object.
- Execution of the iterator block is suspended. If the `yield return` statement is within one or more `try` blocks, the associated `finally` blocks are *not* executed at this time.
- The `MoveNext` method of the enumerator object returns `true` to its caller, indicating that the enumerator object successfully advanced to the next item.

The next call to the enumerator object's `MoveNext` method resumes execution of the iterator block from where it was last suspended.

A `yield break` statement is executed as follows:

- If the `yield break` statement is enclosed by one or more `try` blocks with associated `finally` blocks, control is initially transferred to the `finally` block of the innermost `try` statement. When and if control reaches the end point of a `finally` block, control is transferred to the `finally` block of the next enclosing `try` statement. This process is repeated until the `finally` blocks of all enclosing `try` statements have been executed.
- Control is returned to the caller of the iterator block. This is either the `MoveNext` method or `Dispose` method of the enumerator object.

Because a `yield break` statement unconditionally transfers control elsewhere, the end point of a `yield break` statement is never reachable.

22.4.1 Definite assignment

For a `yield return` statement *stmt* of the form:

```
yield return expr ;
```

- A variable *v* has the same definite assignment state at the beginning of *expr* as at the beginning of *stmt*.
- If a variable *v* is definitely assigned at the end of *expr*, it is definitely assigned at the end point of *stmt*; otherwise; it is not definitely assigned at the end point of *stmt*.

22.5 Implementation example

This section describes a possible implementation of iterators in terms of standard C# constructs. The implementation described here is based on the same principles used by the Microsoft C# compiler, but it is by no means a mandated implementation or the only one possible.

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

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

```
   public void Push(T item) {
      if (items == null) {
         items = new T[4];
      }
      else if (items.Length == count) {
         T[] newItems = new T[count * 2];
         Array.Copy(items, 0, newItems, 0, count);
         items = newItems;
      }
      items[count++] = item;
   }

   public T Pop() {
      T result = items[--count];
      items[count] = T.default;
      return result;
   }

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

The `GetEnumerator` method can be translated into an instantiation of a compiler-generated enumerator class that encapsulates the code in the iterator block, as shown in the following.

```
class Stack<T>: IEnumerable<T>
{
   ...

   public IEnumerator<T> GetEnumerator() {
      return new __Enumerator1(this);
   }

   class __Enumerator1: IEnumerator<T>, IEnumerator
   {
      int __state;
      T __current;
      Stack<T> __this;
      int i;

      public __Enumerator1(Stack<T> __this) {
         this.__this = __this;
      }

      public T Current {
         get { return __current; }
      }

      object IEnumerator.Current {
         get { return __current; }
      }
```

```
class Test
{
   ...
   static IEnumerable<int> FromTo(int from, int to) {
      return new __Enumerable1(from, to);
   }
   class __Enumerable1:
      IEnumerable<int>, IEnumerable,
      IEnumerator<int>, IEnumerator
   {
      int __state;
      int __current;
      int __from;
      int from;
      int to;
      int i;
      public __Enumerable1(int __from, int to) {
         this.__from = __from;
         this.to = to;
      }
      public IEnumerator<int> GetEnumerator() {
         __Enumerable1 result = this;
         if (Interlocked.CompareExchange(ref __state, 1, 0) != 0) {
            result = new __Enumerable1(__from, to);
            result.__state = 1;
         }
         result.from = result.__from;
         return result;
      }
      IEnumerator IEnumerable.GetEnumerator() {
         return (IEnumerator)GetEnumerator();
      }
      public int Current {
         get { return __current; }
      }
      object IEnumerator.Current {
         get { return __current; }
      }
      public bool MoveNext() {
         switch (__state) {
         case 1:
            if (from > to) goto case 2;
            __current = from++;
            __state = 1;
            return true;
         case 2:
            __state = 2;
            return false;
         default:
            throw new InvalidOperationException();
         }
      }
      public void Dispose() {
         __state = 2;
      }
```

```
            void IEnumerator.Reset() {
               throw new NotSupportedException();
            }
         }
      }
```

The enumerable class implements both the enumerable interfaces and the enumerator interfaces, enabling it to serve as both an enumerable and an enumerator. The first time the `GetEnumerator` method is invoked, the enumerable object itself is returned. Subsequent invocations of the enumerable object's `GetEnumerator`, if any, return a copy of the enumerable object. Thus, each returned enumerator has its own state and changes in one enumerator will not affect another. The `Interlocked.CompareExchange` method is used to ensure thread-safe operation.

The `from` and `to` parameters are turned into fields in the enumerable class. Because `from` is modified in the iterator block, an additional `__from` field is introduced to hold the initial value given to `from` in each enumerator.

The `MoveNext` method throws an `InvalidOperationException` if it is called when `__state` is `0`. This protects against use of the enumerable object as an enumerator object without first calling `GetEnumerator`.

Chapter

23. Partial Types

23.1 Partial declarations

A new type modifier, `partial`, is used when defining a type in multiple parts. To ensure compatibility with existing programs, this modifier is different than other modifiers: like `get` and `set`, it is not a keyword, and it must appear immediately before one of the keywords `class`, `struct`, or `interface`.

class-declaration:
attributes$_{opt}$ *class-modifiers*$_{opt}$ `partial`$_{opt}$ `class` *iden*

23.1.2 Modifiers

When a partial type declaration includes an accessibility specification (the `public`, `protected`, `internal`, and `private` modifiers) it must agree with all other parts that include an accessibility specification. If no part of a partial type includes an accessibility specification, the type is given the appropriate default accessibility (§3.5.1).

If one or more partial declarations of a nested type include a `new` modifier, no warning is reported if the nested type hides an inherited member (§3.7.1.2).

If one or more partial declarations of a class include an `abstract` modifier, the class is considered abstract (§10.1.1.1). Otherwise, the class is considered non-abstract.

If one or more partial declarations of a class include a `sealed` modifier, the class is considered sealed (§10.1.1.2). Otherwise, the class is considered unsealed.

Note that a class cannot be both abstract and sealed.

When the `unsafe` modifier is used on a partial type declaration, only that particular part is considered an unsafe context (§18.1).

23.1.3 Type parameters and constraints

If a generic type is declared in multiple parts, each part must state the type parameters. Each part must have the same number of type parameters, and the same name for each type parameter, in order.

When a partial generic type declaration includes type parameter constraints (`where` clauses), the constraints must agree with all other parts that include constraints. Specifically, each part that includes constraints must have constraints for the same set of type parameters, and for each type parameter the set of class, interface, and constructor constrains must be the same. If no part of a partial generic type specifies type parameter constraints, the type parameters are considered unconstrained.

The example

```
partial class Dictionary<K,V>
   where K: IComparable<K>
   where V: IKeyProvider<K>, IPersistable
{
   ...
}

partial class Dictionary<K,V>
   where V: IPersistable, IKeyProvider<K>
   where K: IComparable<K>
{
   ...
}

partial class Dictionary<K,V>
{
   ...
}
```

is correct because those parts that include constrains (the first two) effectively specify the same set of class, interface, and constructor constraints for the same set of type parameters, respectively.

23.1.4 Base class

When a partial class declaration includes a base class specification it must agree with all other parts that include a base class specification. If no part of a partial class includes a base class specification, the base class becomes `System.Object` (§10.1.2.1).

23.1.5 Base interfaces

The set of base interfaces for a type dec

23.2 Name binding

Although each part of an extensible type must be declared within the same namespace, the parts are typically written within different namespace declarations. Thus, different `using` directives (§9.3) may be present for each part. When interpreting simple names (§7.5.2) within one part, only the `using` directives of the namespace declaration(s) enclosing that part are considered. This may result in the same identifier having different meanings in different parts:

```
namespace N
{
   using List = System.Collections.ArrayList;
   partial class A
   {
      List x;           // x has type System.Collections.ArrayList
   }
}
namespace N
{
   using List = Widgets.LinkedList;
   partial class A
   {
      List y;           // y has type Widgets.LinkedList
   }
}
```