C

Version 2.0 Specification

July 2003

Table of Contents

Chapter

19. Introduction to C# 2.0

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

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

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

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

19.1 Generics

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

19.1.1 Why generics?

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

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

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

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

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

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

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

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

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

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

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

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

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

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

19.1.2 Creating and using generics

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

19.1.3 Generic type instantiations

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

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

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

19.1.4 Constraints

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

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

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

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

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

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

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

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

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

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

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

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



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

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

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

19.1.5 Generic methods

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

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

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

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

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

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

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

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

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

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

19.2 Anonymous methods

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
using System;

delegate double Function(double x);

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

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

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

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

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

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

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

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

19.2.1 Method group conversions

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

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

can instead be written

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

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

19.3 Iterators

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

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

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

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

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

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

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

```
using System.Collections.Generic;

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

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

   public T Pop() {...}

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

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

```
using System;

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

The output of the example is:

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

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

```
using System.Collections.Generic;

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

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

   public T Pop() {...}

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

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

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

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

```
using System;
```

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

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

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

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

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

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

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

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

The output of the example is:

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

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

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

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

