9 A Preview of .NET 2.0

Microsoft .NET is an evolving system that is continuously improved and extended. In late 2003 Microsoft announced .NET 2.0 (codename *Whidbey*) as a major new version of .NET; a beta release of it will be available in spring 2004. Version 2.0 will offer a wealth of new features in almost every part of .NET. There are language extensions to C# and the other .NET languages, new library classes, as well as improved functionality for ADO.NET, ASP.NET and web services.

At the time when this book was written, .NET 2.0 was only available as an alpha release. Since there are probably many details that will change before the final release, we decided not to integrate the new features into the regular chapters of our book but instead to provide you with a separate preview chapter that shows you what to expect from .NET in the future.

9.1 New Features in C#

The major new features in C# 2.0 are *generics*, *anonymous methods*, *iterators* and *partial types*. We will have a look at them now.

9.1.1 Generics

In many cases a class should be able to work with arbitrary types of data. The common solution to this is to let the class work with elements of type object, to which all data types are compatible. The problem, however, is that the compiler cannot guarantee that those elements are of a particular type. Furthermore, when an element is retrieved from the class one has to apply a type cast in order to convert it from object to its real data type. Generics are a better solution to this problem.

A generic type is a class, struct, interface or delegate that is parameterized with one or more other types. The type parameters are *placeholders* for concrete types such as int or string that are provided later.

Let us look at an example. If we want to implement a generic Buffer class, we could do it like this:

```
class Buffer<Element> {
    private Element[] data = ...;
    public void Put(Element x) {...}
    public Element Get() {...}
}
```

Element is a placeholder that is written in angle brackets after the class name. It can be used like an ordinary type in the declaration of the data array or the parameters of Put and Get.

When Buffer is used in a declaration (i.e., when it is *instantiated*) its placeholder must be substituted by a real type, for example:

```
Buffer<int> intBuf = new Buffer<int>();
intBuf.Put(3);
int x = intBuf.Get();
```

This example declares a buffer whose elements are statically defined to be of type int. This means that the compiler can check that all values that are passed to Put are of type int. Likewise, the method Get returns an int value (not an object value that has to be cast to an int value first). This not only makes the program more readable but also more efficient, since type casts become unnecessary. Of course, we could also use Buffer to declare a buffer of strings:

```
Buffer<string> stringBuf = new Buffer<string>();
stringBuf.Put("John");
string s = stringBuf.Get();
```

Again, the compiler makes sure that only strings are passed to Put and it knows that the value returned by Get is a string.

Not only classes but also structs, interfaces and delegates can be declared to be generic by parameterizing them with a placeholder type:

```
struct Stack<ElemType> {...}
interface ITokenReader<TokenType> {...}
delegate bool Matches<T>(T value);
```

Constraints

Our class Buffer just stores the elements in the data array but does not perform any operations on them. If the elements are to be compared, however, or if some methods are to be called on them, the compiler must know which operations can be applied to Element variables. This can be specified with a *constraint* that declares the placeholder to be of some minimum type.

Assume that we want to have a class OrderedBuffer which stores a collection of elements sorted by priorities. The priorities can be of any type but they must be comparable. Thus we have to declare OrderedBuffer with a constraint on the placeholder Priority:

```
class OrderedBuffer<Element, Priority> where Priority: IComparable {
    const int size = 100;
    Element[] data = new Element[size];
    Priority[] prio = new Priority[size];
    int lastElem = -1;

    public void Put(Element x, Priority p) { // insert x and p sorted by priority
        int i = lastElem;
        while (i >= 0 && p.CompareTo(prio[i]) > 0) {
            data[i+1] = data[i]; prio[i+1] = prio[i];
            i--;
        }
        data[i+1] = x; prio[i+1] = p;
        lastElem++;
    }

    public void Get(out Element x, out Priority p) {
        x = data[lastElem]; p = prio[lastElem]; lastElem--;
    }
}
```

The where clause in the header of the class specifies that the placeholder Priority must implement the interface IComparable. Therefore the compiler knows that the method CompareTo can be applied to the parameter p in the method Put. If OrderedBuffer is instantiated in a declaration like this

```
OrderedBuffer<string, int> buf = new OrderedBuffer<string, int>();
```

the compiler checks that the type that is substituted for Priority implements the interface IComparable (which is the case for int). We can use the variable buf now to enter data and priorities:

```
buf.Put("network", 10);
buf.Put("printer", 5);
string device; int priority;
buf.Get(out device, out priority);   // device == "network", priority == 10
```

One can also specify multiple constraints for the placeholders:

```
class OrderedBuffer<Element, Priority>
    where Element: MyBaseClass
    where Priority: IComparable, ISerializable {
    ...
}
```

In this example the type substituted for Element must be derived from a class MyBaseClass and the type substituted for Priority must implement the interfaces IComparable and ISerializable.

In addition to classes and interfaces a constraint can also specify a *constructor clause* which is written as new(), for example:

```
class Buffer<Element> where Element: ISerializable, new() {...}
```

This means that the placeholder must be substituted by a class with a parameterless constructor so that Buffer can create and initialize Element objects.

Generic Types and Inheritance

Like normal classes a generic class can inherit from another class and implement several interfaces. Both the base class and the interfaces can be generic again. Our class Buffer<Element>, for example, could be derived from a class List<Element>, which provides the methods Add and Remove for implementing the buffer:

```
class Buffer<Element>: List<Element> {
    ...
    public void Put(Element x) { this.Add(x); } // Add is inherited from List
}
```

The type that is substituted for Element in Buffer is also substituted for Element in List.

A generic type can inherit from a normal type, from an instantiated generic type, or from a generic type with the same placeholder, so:

```
class A<X>: B {...}        // extending a normal type
class A<X>: B<int> {...}   // extending an instantiation of a generic type
class A<X>: B<X> {...}     // extending a generic type with the same placeholder
```

A normal class, however, can never inherit from a generic type. The following declaration is therefore illegal:

```
class A: B<X> {...}
```

In the same way that classes can be derived from generic types, interfaces can be derived from generic interfaces.

Assignment Compatibility Between Generic Types

Generic subtypes are compatible with their base types. Assume that we have the following declarations:

```
class B<X>: A {...}
class C<X>: B<X> {...}
```

then we can perform the assignments:

```
A a = new B<int>();              // OK, B<int> is a subtype of A
a = new C<string>();             // OK, C<string> is an indirect subtype of A
B<float> b = new C<float>();     // OK, C<float> is a subtype of B<float>
```

However, the assignment

```
B<int> b = new C<short>();
```

is illegal, because C<short> is not a subtype of B<int>. The compiler will report an error.

Overriding Methods in Generic Classes

If a method of an instantiated generic base type is overridden in a subtype, any placeholder that appears as a parameter type in the base method is substituted by the corresponding concrete type:

```
class MyBuffer: Buffer<int> {
   public override void Put(int x) {...}        // Element is substituted by int
}
```

However, if a method from a non-instantiated generic base type is overridden, the placeholder is not substituted:

```
class MyBuffer<Element>: Buffer<Element> {
   public override void Put(Element x) {...} // Element is not substituted
}
```

Run-time Checks

Like any other type, an instantiated generic type (for example, Buffer<int>) can be used in type tests and type casts:

```
Buffer<int> buf = new Buffer<int>();
object obj = buf;                // the dynamic type of obj is Buffer<int>
...
if (obj is Buffer<int>)          // run-time type test; returns true here
   buf = (Buffer<int>) obj;      // type cast
```

Generic types can even be used in reflection, for example:

```
Type t = typeof(Buffer<int>);
Console.WriteLine(t.FullName); // prints Buffer[[System.Int32,...]]
```

Generic Methods

Although methods are not types, they too can be generic, that is, they can be parameterized with other types. In this way one can implement a method that is applicable to data of arbitrary types. Here is an example of a Sort routine that can be used to sort any array whose elements implement the interface IComparable.

```
static void Sort<T>(T[] a) where T: IComparable {
   for (int i = 0; i < a.Length-1; i++) {
      for (int j = i+1; j < a.Length; j++) {
         if (a[j].CompareTo(a[i]) < 0) { T x = a[i]; a[i] = a[j]; a[j] = x; }
      }
   }
}
```

This method can now be used as follows:

```
int[] a = {5, 4, 7, 3, 1};
Sort<int>(a);
```

The C# compiler is even clever enough to derive the type that should be substituted for the placeholder from the type of the parameter a, so we can just write:

```
Sort(a);
```

Notice that we could also have implemented Sort as a non-generic method Sort(object[] a). The difference is that the compiler can check that the parameter of the generic method is an int array while for the non-generic case the compiler cannot even check that the elements of the parameter array support the interface IComparable.

Null Values

For any placeholder T there is a value T.default that denotes the null value of the corresponding concrete type:

```
class A<T> {
   T a = 0;            // illegal; the compiler reports an error
   T b = null;         // illegal; the compiler reports an error
   T c = T.default;    // OK; assigns 0, null, false, or '\0' depending on the concrete type
}
```

If x is declared with a placeholder type T the check

```
if (x == null) ...
```

does a null check if T was substituted by a reference type, otherwise it yields false.

How Generic Types are Implemented

If we declare a generic class Buffer<Element> the compiler generates CIL code that serves as a template for later instantiations. When the class is instantiated the JIT compiler creates code for different concrete classes depending on whether the placeholder was substituted by a value type or by a reference type.

For every value type V that is substituted for the placeholder Element the JIT compiler creates a concrete class Buffer<V>, whereas all reference types R that are substituted for Element share a common implementation of the class Buffer<R>:

```
Buffer<int> a = new Buffer<int>();              // creates a class Buffer<int>
Buffer<int> b = new Buffer<int>();              // reuses Buffer<int>
Buffer<float> c = new Buffer<float>();          // creates a class Buffer<float>
Buffer<string> d = new Buffer<string>();        // creates a class Buffer<refType>
Buffer<Person> e = new Buffer<Person>(); // reuses Buffer<refType>
```

9.1.2 Anonymous Methods

Anonymous methods simplify the use of delegates. Let's explain this with an example. Assume that we have a dialog window with a text box and a button. Whenever the button is clicked the numeric value of the text box should be added to a local sum. Thus we install a delegate for the Click event of the button:

```
public class Sample {
    Button button = new Button("Add");
    TextBox amount = new TextBox();
    int sum = 0;

    void AddAmount(object sender, EventArgs e) {
        sum += Convert.ToInt32(amount.Text);
    }
    public void Run() {
        ...
        button.Click += new EventHandler(AddAmount);
    }
}
```

This works fine, but it is slightly inconvenient, since we have to declare a method AddAmount just to execute a single statement in response to a button click. We have to invent a name for this method. Even worse, the method cannot be local to Run and so the variables that are accessed in AddAmount cannot be local either.

Anonymous methods allow us to specify the body of AddAmount right at the place where it is assigned to the Click event:

```
public class Sample {
    public void Run() {
        Button button = new Button("Add");
        TextBox amount = new TextBox;
        int sum = 0;
        button.Click = delegate (object sender, EventArgs e) {
            sum += Convert.ToInt32(amount.Text);
        };
    }
}
```

An anonymous method consists of the keyword delegate, an optional formal parameter list and a method body. No name has to be invented for such a method and its body can access the local variables of the enclosing method Run. If the formal parameters are not used in the body of the anonymous method (as in our example) they can be omitted, so we can just write:

```
button.Click = delegate { sum += Convert.ToInt32(amount.Text); };
```

While speaking about delegates it is worth mentioning that the creation of delegate objects is simplified in C# 2.0. Instead of having to write

```
button.Click += new EventHandler(AddAmount);
```

we can just write

```
button.Click += AddAmount;
```

The compiler will derive the type of the delegate object that is to be created from the type of the Click event.

9.1.3 Iterators

The foreach loop is a convenient way to iterate over collections. In order to be able to do this, however, the collection has to implement the interface IEnumerable from the namespace System.Collections. This requires the methods MoveNext and Reset to be implemented, as well as the property Current (see Section 4.1.1). C# 2.0 offers a simpler solution for that. If a class has a method

```
public IEnumerator GetEnumerator() {...}
```

that contains one or more *yield statements* then it is possible to apply a foreach statement to objects of this class. Let’s look at an example. Assume that we have a class Store that maintains a set of fruits. If we want to iterate over these fruits we have to implement a method GetEnumerator as follows:

```
class Store {
    Fruit apple = new Fruit("apple", ...);
    Fruit orange = new Fruit("orange", ...);
    Fruit banana = new Fruit("banana", ...);

    public IEnumerator GetEnumerator() {
        yield return apple;
        yield return orange;
        yield return banana;
    }
}
```

The yield statements return a sequence of fruits and the foreach loop can be used to iterate over them:

```
Store store = new Store();
foreach (Fruit fruit in store) Console.WriteLine(fruit.name); // apple, orange, banana
```

Note that the class Store no longer has to implement IEnumerable but just provides a GetEnumerator method, which is automatically transformed into a method that returns an object of a compiler-generated class _Enumerator:

```
class _Enumerator: IEnumerator {
    public object Current { get {...} }
    public bool MoveNext() {...}
    public void Dispose() {...}
}
```

The above foreach loop is then translated to:

```
IEnumerator _e = store.GetEnumerator();
try {
    while (_e.MoveNext()) Console.WriteLine(((Fruit)_e.Current).name);
} finally {
    if (_e != null) _e.Dispose();
}
```

Every call to MoveNext runs to the next yield statement, which stores the yielded value in a private field from where it can be retrieved by Current.

Instead of IEnumerator, the method GetEnumerator should preferably use the return type IEnumerator<Fruit> (declared in System.Collections.Generic). In this way the property Current will return a value of type Fruit instead of type object and the type cast in the translated form of the foreach loop can be omitted.

The yield statement comes in two forms. The statement

```
yield return expression;
```

yields a value for the next iteration of the foreach loop. The type of *expression* must be T if the return type of GetEnumerator is IEnumerator<T>, otherwise a value of type object is returned. The statement

```
yield break;
```

terminates the foreach loop and does not yield a value.

Specific Iterators

In addition to the standard iterator GetEnumerator, a class can also implement any number of more specific iterators, which are methods or properties with the type IEnumerable (or IEnumerable<T>) that contain a yield statement. The following class demonstrates this by providing three ways for iterating over a private data array:

```
class MyList {
    int[] data = ...;

    public IEnumerator<int> GetEnumerator() {          // standard iterator
        for (int i = 0; i < data.Length; i++) yield return data[i];
    }

    public IEnumerable<int> Range(int from, int to) {   // specific iterator method
        for (int i = from; i <= to; i++) yield return data[i];
    }

    public IEnumerable<int> Downwards {                 // specific iterator property
        get {
            for (int i = data.Length-1; i >= 0; i--) yield return data[i];
        }
    }
}
```

Note that specific iterators return IEnumerable<T> while a standard iterator returns IEnumerator<T>. These iterators can now be used as follows:

```
MyList list = new MyList();
foreach (int x in list) Console.WriteLine(x);
foreach (int x in list.Range(2, 7)) Console.WriteLine(x);
foreach (int x in list.Downwards) Console.WriteLine(x);
```

For those who are interested in the implementation of all this: The method Range is transformed into a method that returns a compiler-generated object of type IEnumerable<int>, which has a GetEnumerator method that returns a compiler-generated object of type IEnumerator<int>, which is then used in the foreach loop as explained above. Quite tricky, isn't it? But as a programmer you don't have to care about it.

9.1.4 Partial Types

Classes, structs and interfaces are usually implemented in a single file, which is good, because it promotes readability and maintainability. In some cases, however, it makes sense to split up a type's implementation into several files. In C# 2.0 this can be done by declaring a type with the modifier partial.

The following example shows a type C whose source code is split into two parts that are implemented in the files Part1.cs and Part2.cs.

```
// this code is in file Part1.cs
public partial class C {
    int x;
    public void M1() {...}
}
```

```
// this code is in file Part2.cs
public partial class C {
    string y;
    public void M2() {...}
}
```

Although the separation of a type into several files should be the exception it can have advantages occasionally:

- ❑ The members of a type can be grouped according to their functionality.
- ❑ Several developers can work on a type concurrently because each of them is using his own file.
- ❑ Parts of a type can be generated by a program while others can be written by hand. In this way it is easier to separate machine-generated parts from hand-written ones. If the generated parts are re-created they don't have to be merged with the hand-written parts.

9.2 New Features in the Base Class Library

The Base Class Library (BCL) provides a rich set of types for rapid application development. Under .NET 2.0 some weaknesses of this library were removed and new functionality was added. Existing features were improved (for example, better graphics performance, simplified file access, and a more powerful class Console) and new features were added (for example, a web browser control for Windows forms, as well as new namespaces for generic collections and for data protection). In this section we will go through a few of the most important new features.

9.2.1 Generic Collections

The new namespace System.Collections.Generic provides support for generic collections, which offer a better performance for strongly typed elements than traditional collections do. The most important classes of this namespace are List<T>, Stack<T>, Queue<T>, Dictionary<T, U> and SortedDictionary<T, U>. They offer the same methods as the traditional collection classes with the exception that all elements in the collections are strongly typed. The following example shows the creation and use of a List (which is the generic counterpart of ArrayList):

```
// create a list of strings
List<string> list = new List<string>();
// add three strings
list.Add("Mike");
list.Add("Andrew");
list.Add("Susan");
```