C# 2.0

1. Generics
2. Nullable types
3. Iterators
4. yield Statement
5. Partial Classes/Structs/Interfaces
6. Anonymous methods
7. The :: operator (Namespace Alias Qualifier)
8. Static Classes
9. Covariance and contravariance
10. Fixed-size buffers

Note: fixed keyword [only with built-in struct except decimal, only in unsafe, cannot be static]

1. Friend assemblies
2. Extern Aliases [extern keyword has been available since C# 1.0]
3. Delegate Method Group Conversion
4. Access Modifiers with Accessors [Note: For properties and indexers]
5. #pragma directives
6. ?? Operator or Null-Coalesce Operator
7. System.Predicate
8. System.Action

**1. Generics**

At its core, the term generics means parameterized types. Parameterized types are important because they enable you to create classes, structures, interfaces, methods, and delegates in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data. A class, structure, interface, method, or delegate that operates on a parameterized type is called generic, as in generic class or generic method.

It is important to understand that C# has always given you the ability to create generalized code by operating through references of type object. Because object is the base class of all other classes, an object reference can refer to any type of object. Thus, in pre-generics code, generalized code used object references to operate on a variety of different kinds of objects.

**2. Nullable Types**

C# 2.0 added a new feature that provides an elegant solution to what is both a common and irritating problem. The feature is the nullable type. The problem is how to recognize and handle fields that do not contain values (in other words, unassigned fields). To understand the problem, consider a simple customer database that keeps a record of the customer’s name, address, customer ID, invoice number, and current balance. In such a situation, it is possible to create a customer entry in which one or more of those fields would be unassigned. For example, a customer may simply request a catalog. In this case, no invoice number would be needed and the field would be unused.

In the past, handling the possibility of unused fields required the use of either placeholder values or an extra field that simply indicated whether a field was in use. Of course, placeholder values could work only if there was a value that would otherwise be invalid, which won’t be the case in all situations. Adding an extra field to indicate if a field is in use works in all cases, but having to manually create and manage such a field is an annoyance. The nullable type solves both problems.

Nullable Basics

A nullable type is a special version of a value type that is represented by a structure. In addition to the values defined by the underlying type, a nullable type can also store the value null. Thus, a nullable type has the same range and characteristics as its underlying type. It simply adds the ability to represent a value that indicates that a variable of that type is unassigned. Nullable types are objects of System.Nullable<T>, where T must be a nonnullable value type.

REMEMBER Only value types have nullable equivalents.

A nullable type can be specified two different ways. First, you can explicitly declare objects of type Nullable<T>, which is defined in the System namespace. For example, this creates int and bool nullable types:

System.Nullable<int> count;

System.Nullable<bool> done;

The second way to declare a nullable type is much shorter and is more commonly used.

Simply follow the type name with a ?. For example, the following shows the more common

way to declare a nullable int and bool type:

int? count;

bool? done;

When using nullable types, you will often see a nullable object created like this:

int? count = null;

Expressions like:

if(count != null) // has a value

if(count.HasValue) // has a value

**3. Iterators**

A feature related to an enumerator is the iterator. It simplifies the process of creating classes, such as custom collections, that can be cycled through by a foreach loop.

Using Iterators

Implementing IEnumerator and IEnumerable can be made easier through the use of an iterator. An iterator is a method, operator, or accessor that returns the members of a set of objects, one member at a time, from start to finish. For example, assuming some array that has five elements, then an iterator for that array will return those five elements, one at a time. Implementing an iterator is another way to make it possible for an object of a class to be used in a foreach loop.

**4. yield Statement**

C# 2.0 adds the yield statement for creating enumerators easily. yield return returns one element of a collection and moves the position to the next element, and yield break stops the iteration.

**5. Partial Classes/Structs/Interfaces**

Beginning with C# 2.0, a class, structure, or interface definition can be broken into two or more pieces, with each piece residing in a separate file. This is accomplished through the use of the partial contextual keyword. When your program is compiled, the pieces are united.

Note that it is legal to have partial generic classes. However, the type parameters of each partial declaration must match the other parts.

**6. Anonymous Methods**

C# 2.0 added the ability to pass an anonymous method to a delegate. An anonymous method is, essentially, a block of code that is passed to a delegate. The main advantage to using an anonymous method is simplicity. In many cases, there is no need to actually declare a separate method whose only purpose is to be passed to a delegate. In this situation, it is easier to pass a block of code to the delegate than it is to first create a method and then pass that method to the delegate.

An anonymous method is one way to create an unnamed block of code that is associated with a specific delegate instance. An anonymous method is created by following the keyword delegate with a block of code.

**7. The :: operator (Namespace Alias Qualifier)**

Although namespaces help prevent name conflicts, they do not completely eliminate them. One way that a conflict can still occur is when the same name is declared within two different namespaces, and you then try to bring both namespaces into view. For example, assume that two different namespaces contain a class called MyClass. If you attempt to bring these two namespaces into view via using statements, MyClass in the first namespace will conflict with MyClass in the second namespace, causing an ambiguity error. In this situation, you can use the :: namespace alias qualifier to explicitly specify which namespace is intended. The :: namespace alias qualifier is a new feature added by C# 2.0.

The :: operator has this general form:

namespace-alias::identifier

Here, namespace-alias is the name of a namespace alias, and identifier is the name of a member of that namespace.

To understand why the namespace alias qualifier is needed, consider the following program. It creates two namespaces, Counter and AnotherCounter, and both declare a class called CountDown. Furthermore, both namespaces are brought into view by using statements. Finally, an attempt is made to instantiate an object of type CountDown.

using Ctr = Counter;

Then, this alias is used to qualify CountDown, as shown here:

Ctr::CountDown cd1 = new Ctr::CountDown();

global::CountDown cd2 = new global::CountDown();

Note the following has been available since the creation of C#.

Counter.CountDown cd1 = new Counter.CountDown ();

AnotherCounter.CountDown cd1 = new AnotherCounter.CountDown ();

**8. Static Classes**

C# 2.0 added the ability to create static classes. There are two key features of a static class. First, no object of a static class can be created. Second, a static class must contain only static members. The main benefit of declaring a class static is that it enables the compiler to prevent any instances of that class from being created. Thus, the addition of static classes does not add any new functionality to C#. It does, however, help prevent errors.

Within the class, all members must be explicitly specified as static. Making a class static does not automatically make its members static.

The main use of a static class is to contain a collection of related static methods.

**9. Covariance and Contravariance**

Two other new delegate-related features added by C# 2.0 are covariance and contravariance. Normally, the method that you pass to a delegate must have the same return type and signature as the delegate. However, covariance and contravariance relax this rule slightly, as it pertains to derived types. Covariance enables a method to be assigned to a delegate when the method’s return type is a class derived from the class specified by the return type of the delegate. Contravariance enables a method to be assigned to a delegate when a method’s parameter type is a base class of the class specified by the delegate’s declaration.

**10. Fixed-size buffers [Note: Available only in an unsafe context]**

C# 2.0 expanded the use of the fixed keyword to enable you to create fixed-size, onedimensional arrays. In the C# documentation, these are referred to as fixed-size buffers. A fixed-size buffer is always a member of a struct. The purpose of a fixed-size buffer is to allow the creation of a struct in which the array elements that make up the buffer are contained within the struct. Normally, when you include an array member in a struct, only a reference to the array is actually held within the struct. By using a fixed-size buffer, you cause the entire array to be contained within the struct. This results in a structure that can

be used in situations in which the size of a struct is important, such as in mixed-language programming, interfacing to data not created by a C# program, or whenever a non-managed struct containing an array is required. Fixed-size buffers can only be used within an unsafe context.

**Sizeof [Note: Available in both unsafe and safe context]**

When working in an unsafe context, you might occasionally find it useful to know the size, in bytes, of one of C#’s value types. To obtain this information, use the sizeof operator.

In general, sizeof is intended primarily for special-case situations, especially when working with a blend of managed and unmanaged code.

**11. Friend Assemblies**

C# 2.0 added the ability to make one assembly the friend of another. A friend has access to the internal members of the assembly of which it is a friend. This feature makes it possible to share members between selected assemblies without making those members public. To declare a friend assembly, you must use the InternalsVisibleTo attribute.

**12. Extern Aliases**

A second form of extern was added by C# 2.0. An extern provides an alias for an external assembly. It is used in cases in which a program includes two separate assemblies that both contain the same type name.

For example, if an assembly called test1 contains a class called MyClass and test2 also contains a class called MyClass, then a conflict will arise if both classes need to be used within the same program. To solve this problem, you must create an alias for each assembly. This is a two-step process.

First, you must specify the aliases using the /r compiler option.

For example:

/r:Asm1=test1.dll

/r:Asm2=test2.dll

Second, you must specify extern statements that refer to these aliases. Here is the form of

extern that creates an assembly alias:

extern alias assembly-name;

Continuing the example, these lines must appear in your program:

extern alias Asm1;

extern alias Asm2;

**13. Delegate Method Group Conversion**

C# 2.0 added an option that significantly simplifies the syntax that assigns a method to a delegate. This feature is called method group conversion, and it allows you simply to assign the name of a method to a delegate, without the use of new or explicitly invoking the delegate’s constructor.

**14. Access Modifiers with Accessors [Note: For properties and indexers]**

Beginning with C# 2.0, you can specify an access modifier, such as private, when declaring a get or set accessor. Doing so enables you to control access to an accessor. For example, you might want to make set private to prevent the value of a property or an indexer from being set by code outside its class. In this case, the property or indexer could be read by any code, but set only by a member of its class.

**15. #pragma directives**

The #pragma directive, added by C# 2.0, is used to give instructions, such as specifying an option, to the compiler.

There are two options supported by #pragma. The first is warning, which is used to enable or disable specific compiler warnings. It has these two forms:

#pragma warning disable warnings

#pragma warning restore warnings

Here, warnings is a comma-separated list of warning numbers. To disable a warning, use the disable option. To enable a warning, use the restore option.

For example, this #pragma statement disables warning 168, which indicates when a variable is declared but not used:

#pragma warning disable 168

The second #pragma option is checksum. It is used to generate checksums for ASP.NET projects. It has this general form:

#pragma checksum “filename” “{GUID}” “check-sum”

Here, filename is the name of the file, GUID is the globally unique identifier associated with filename, and check-sum is a hexadecimal number that contains the checksum. This string must contain an even number of digits.

**16. ?? Operator or Null-Coalesce Operator**

If you attempt to assign a nullable object to a variable of its underlying type, a System.InvalidOperationException will be thrown if the nullable object contains a null value. You can avoid this possibility by specifying a default value that will be assigned to the non-nullable variable when the nullable object contains null. To do so, use the ?? operator, which is called the null coalescing operator. It has this general form:

nullable-object ?? default-value

If nullable-object contains a value, then the value of the ?? operation is that value. Otherwise, the value of the ?? operation is default-value.

For example, in the following code balance is null. This causes currentBalance to be assigned the value 0.0 and no exception will be thrown.

double? balance = null;

double currentBalance;

currentBalance = balance ?? 0.0;

In the next sequence, balance is given the value 123.75:

double? balance = 123.75;

double currentBalance;

currentBalance = balance ?? 0.0;

Now, currentBalance will contain the value of balance, which is 123.75.

One other point: the right-hand expression of the ?? is evaluated only if the left-hand expression does not contain a value.

**17. System.Predicate**

System.Predicate that returns either true or false, based upon some condition.

It is declared as shown here:

public delegate bool Predicate<T> (T obj)

The object to be tested against the condition is passed in obj. If obj satisfies that condition, the predicate must return true. Otherwise, it must return false. Predicates are used by several new methods added to Array by C# 2.0, including Exists( ), Find( ), FindIndex( ), and FindAll( ).

**18. System.Action**

Another new feature introduced by C# 2.0 is the System.Action delegate. An Action is used by another new C# feature, Array.ForEach( ), to perform an action on each element of an array. Action is declared as shown here:

public delegate void Action<T> (T obj)

The object to be acted upon is passed in obj. When used with ForEach( ), each element of the array is passed to obj in turn. Thus, through the use of ForEach( ) and Action, you can, in a single statement, perform an operation over an entire array.

The following program demonstrates both ForEach( ) and Action. It first creates an array of MyClass objects, then uses the method show( ) to display the values. Next, it uses neg( ) to negate the values. Finally, it uses show( ) again to display the negated values. These operations all occur through calls to ForEach( ).