# 14 Unsafe code, exceptions, anonymous methods, iterators

We will see that C# allows suspending the verification of code by the CLR to allow developers to directly access memory using pointers. Hence with C#, you can complete, in a standard way, certain optimizations which were only possible within unmanaged development environments such as C++. These optimizations concern, for example, the processing of large amounts of data in memory such as bitmaps.

As with the C++ and Java languages, C# exposes a simple yet powerful exception management system.

We will finally see that the C#2 language adds two syntax features which are close to functional programming which, in certain specific cases, can significantly increate code readability.

## Pointers and unsafe code

**C++ › C#:** C++ does not know the notion of code management. This is one of the advantages of C++ as it allows the use of pointers and thus allows developers to write optimized code which is closer to the target machine.

This is also a disadvantage of C++ since the use of pointers is cumbersome and potentially dangerous, significantly increasing the development effort and maintenance required.

**C#:** Before the .NET platform, 100% of the code executed on the *Windows* operating system was unmanaged. This means the executable contains the code directly in machine instructions which are compatible with the type of processor (i.e. machine language code). The introduction of the managed execution mode with the .NET platform is revolutionary. The main sources of hard to track bugs are detected and resolved by the CLR. Amongst these:

- Array access overflows (Now dynamically managed by the CLR).
- *Memory leaks* (Now mostly managed by the garbage collector).
- The use of an invalid pointer. This problem is solved in a radical way as the manipulation of pointers if forbidden in managed mode.

However, during the presentation of the CTS at page 285, we have shown that the CLR knows how to manipulate three kinds of pointers:

- *Managed pointers*. These pointers can point to data contained in the object heap managed by the garbage collector. These pointers are not used explicitly by the C# code. They are thus used implicitly by the C# compiler when it compiles methods with `out` and `ref` arguments.
- *Unmanaged function pointers*. The section at page 224 discusses the use of these pointers.
- *Unmanaged pointers*. These pointers can point to any data contained in the user addressing space of the process. The C# language allows to use this type of pointers in zones of code considered *unsafe*. The IL code emitted by the C# compiler corresponding to the zones of code which use these unmanaged pointers make use of specialized IL instructions. Their effect on

the memory of the process cannot be verified by the JIT compiler of the CLR. Consequently, a malicious user can take advantage of unsafe code regions to accomplish malicious actions. To counter this weakness, the CLR will only allow the execution of this code at run-time if the code has the `SkipVerification` CAS meta-permission

Since it allows to directly manipulating the memory of a process through the use of an unmanaged pointer, unsafe code is particularly useful to optimize certain processes on large amounts of data stored in structures. An example of optimization of image manipulation, using pointers can be found at page 580.

### Compilation options to allow unsafe code

Unsafe code must be used on purpose and you must also provide the `/unsafe` option to the `csc.exe` compiler to tell it that you are aware that the code you wish to compile contains zones which will be seen as unverifiable by the JIT compiler. *Visual Studio* offers the *Build › Allow unsafe code* project property to indicate that you wish to use this compiler option.

### Declaring unsafe code in C#

In C#, the `unsafe` keyword lets the compiler know when you will use unsafe code. It can be used in three situations:

- Before the declaration of a class or structure. In this case, all the methods of the type can use pointers.
- Before the declaration of a method. In this case, the pointers can be used within the body of this method and in its signature.
- Within the body of a method (static or not). In this case, pointers are only allowed within the marked block of code. For example:

```
unsafe{
   ...
}
```

Let us mention that if a method accepts at least one pointer as an argument or as a return value, the method (or its class) must be marked as unsafe, but also all regions of code calling this method must also be marked as unsafe.

## Using pointers in C#

**C++ › C#:** The use syntax for pointers is identical in C# as in C++ except for certain particular pointers: in C#, the `int *p1,p2;` declaration makes it so `p1` is a pointer on an integer and `p2` is an integer.

Only certain types can be used as pointers.

In C#, it is necessary to pin objects in memory to use pointers on them.

**C#:** Each object, whether it is a value or reference type instance, has a memory address at which it is physically located in the process. This address is not necessarily constant during the lifetime of the object as the garbage collector can physically move objects store in the heap.

### .NET types that support pointers

For certain types, there is a dual type, the unmanaged pointer type which corresponds to the managed type. A pointer variable is in fact the address of an instance of the concerned type. The set of types which authorizes the use of pointers limits itself to all value types, with the exception of structures with at least one reference type field. Consequently, only instances of the following types can be used through pointers: primitive types; enumerations; structures with no reference type fields; pointers.

## Declaring pointers

A pointer might point to nothing. In this case, it is **extremely important** that its value should be set to null (0). In fact, the majority of bugs due to pointers come from pointers which are not null but which point to invalid data.

The declaration of a pointer on the `FooType` is done as follows:

```
FooType * pointeur;
```

For example:

```
long * pAnInteger = 0;
```

Note that the declaration…

```
int * p1,p2;
```

…makes it so that `p1` is a pointer on an integer and `p2` is a pointer.

## Dereferencing and indirection operators

In C#, we can obtain a pointer on a variable by using the *address of* operator &. For example:

```
long anInteger = 98;
long * pAnInteger = &anInteger;
```

We can access to the object through the *indirection* operator `*`. For example:

```
long anInteger = 98;
long * pAnInteger = &anInteger;
long anAnotherInteger = *pAnInteger;
// Here, the value of 'anAnotherInteger' is 98.
```

## The sizeof operator

The `sizeof` operator allows obtaining the size in bytes of instances of a value type. This operator can only be used in unsafe mode. For example:

```
int i = sizeof(int)     // i is equal to 4
int j = sizeof(double)  // j is equal to 8
```

## Pointer arithmetic

A pointer on a type `T` can be modified through the use of the `'++'` and `'--'` unary operator. The '–' operator can also be used with pointers.

- The `'++'` operator increments the pointer by `sizeof(T)` bytes.
- The `'--'` operator decrements the pointer by `sizeof(T)` bytes.
- The `'-'` operator used between two pointers of same type `T`, returns a value of type `long`. This value is equal to the byte offset between the two pointers divided by `sizeof(T)`.

The comparison can also be used on two pointers of a same or different type. The supported comparison operators are:

```
==        !=        <        >        <=        >=
```

## Pointer casting

Pointers in C# do not derive from the `Object` class and thus the boxing and unboxing (at page 292) does not exist on pointers. However, pointers support both implicit and explicit casting.

*Implicit casts* are done from any type of pointer to a pointer of type `void*`.

*Explicit casts* are done from:

- Any pointer type to any other pointer type.
- Any pointer type to the `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` types (caution, we are not talking about the `sbyte*`, `byte*`, `short*`... types).
- One of `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong` types to any pointer type.

## Double pointers

Let us mention the possibility of using a pointer on a pointer (although somewhat useless in C#). Here, we talk of a *double pointer*. For example:

```
long aLong = 98;
long * pALong = &aLong;
long ** ppALong = &pALong ;
```

It is important to have a naming convention for pointers and double pointers. In general the name of a pointer is prefixed with `'p'` while the name of a double pointer is prefixed with `'pp'`.

## Pinned object

**C++ › C#:** The notion of object pinning is completely unknown to C++, since it comes from the fact that the .NET platform leaves the management of the managed heap to the garbage collector.

**C#:**

At page 97 we explain how the garbage collector has the possibility of physically moving the objects for which it is responsible. Objects managed by the garbage collector are generally reference type's instances while pointed objects are value type's instances. If a pointer points to a value type field of an instance of a reference type, there will be a potential problem as the instance of the reference type can be moved at any time by the garbage collector. The compiler forces the developer to use the `fixed` keyword in order to tell the garbage collector not to move reference type instances which contain a value field pointed to by a pointer. The syntax of the `fixed` keyword is the following:

*Example 14-1*

```
class Article { public long Price = 0;}
unsafe class Program {
   unsafe public static void Main() {
      Article article = new Article();
      fixed ( long* pPrice = &article.Price ){
         // Here, you can use the pointer 'pPrice' and the object
         // referenced by 'article' cannot be moved by the GC.
      }
      // Here, 'pPrice' is not available anymore and the object
      // referenced by 'article' is not pinned anymore.
   }
}
```

If we had not used the `fixed` keyword in this example, the compiler would have produced an error as it can detect that the object referenced by the `article` may be moved during execution.

We can pin several objects of a same type in the same `fixed` block. If we need to pin objects of a several types, you will need to use nested `fixed` blocks.

You must pin objects the least often as possible, for the shortest duration possible. When objects are pinned, the work of the garbage collector is impaired and less efficient.

Variables of a value type declared as local variable in a method do not need to be pinned since they are not managed by the garbage collector.

## Pointers and arrays

**C++ › C#:** The notion of pointers and arrays are close in C++ from the fact that elements of an array are stored in a contiguous memory area. This similarity also exists in C#.

**C#:** In C#, the elements of an array made from a type which can be pointed to can be accessed by using pointers. Let us precise that an array is an instance of the `System.Array` class and is stored on the managed heap by the garbage collector. Here is an example which both shows the syntax but also the overflow of the array (which is not detected at compilation or execution!) due to the use of pointers:

*Example 14-2*

```
using System;
public class Program {
   unsafe public static void Main() {
      // Create an array of 4 integers.
      int [] array = new int[4];
      for( int i=0; i < 4; i++ )
         array[i] = i*i;
      Console.WriteLine( "Display 6 items (oops!):" );
      fixed( int *ptr = array )
         for( int j = 0; j< 6 ; j++ )
            Console.WriteLine( *(ptr+j) );
      Console.WriteLine( "Display all items:" );
      foreach( int k in array )
         Console.WriteLine(k);
   }
}
```

Here is the display:

```
Display 6 items (oops!):
0
1
4
9
0
2042318948
Display all items:
0
1
4
9
```

Note that it is necessary to only pin the array and not each element of the array. This confirms the fact that during execution, the value type elements of an array are store in contiguous memory.

## Fixed arrays

C#2 allows the declaration of an array field composed of a fixed number of primitive elements within a structure. For this, you simply need to declare the array using the `fixed` keyword and the structure using the `unsafe` keyword. In this case, the field is not of type `System.Array` but of type a pointer to the primitive type (i.e. the `FixedArray` field is of type `int*` in the following example):

*Example 14-3*

```
unsafe struct Foo {
   public fixed int FixedArray[10];
   public int Overflow;
}
unsafe class Program {
   unsafe public static void Main() {
      Foo foo = new Foo();
      foo.Overflow = -1;
      System.Console.WriteLine( foo.Overflow );
      foo.FixedArray[10] = 99999;
      System.Console.WriteLine( foo.Overflow );
   }
}
```

This example displays:

```
-1
99999
```

Understand that `FixedArray[10]` is a reference to the eleventh element of the array since the indexes are zero based. Hence, we assign the 99999 value to the `Overflow` integer.

## Allocating memory on the stack with the stackalloc keyword

**C++ › C#:** C# allows, using a dedicated syntax, the allocation of an array of elements which can be pointed to on the stack. The result is the same as the static allocation of an array in C++.

**C#:** C# allows you to allocate on the stack an array of elements of a type which can by pointed to. The `stackalloc` keyword is used for this, with the following syntax:

*Example 14-4*

```
public class Program {
   unsafe public static void Main(){
      int * array = stackalloc int[100];
      for( int i = 0; i< 100 ; i++ )
         array[i] = i*i;
   }
}
```

None of the elements of the array are initialized, which means that it is the responsibility of the developer to initialize them. If there is insufficient memory on the stack, the `System.StackOverflowException` exception is raised.

The size of the stack is relatively small and we can allocate arrays containing only a few thousand elements. This array is freed implicitly when the method returns.

### Strings and pointers

The C# compiler allows you to obtain a pointer of type `char` from an instance of the `System.String` class. You can use this feature to circumvent managed string immutability. Let us remind that managed string immutability allows to considerably ease their use. However, this can have a negative impact on performance. The `System.StringBuiler` class is not always the proper solution and it can also be useful to directly modify the characters of a string. The following example shows how to use this feature to write a method which converts a string to uppercase:

*Example 14-5*

```
public class Program {
   static unsafe void ToUpper( string str ) {
      fixed ( char* pfixed = str )
         for ( char* p = pfixed; *p != 0; p++ )
            *p = char.ToUpper(*p);
   }
   static void Main() {
      string str = “Hello”;
      System.Console.WriteLine(str);
      ToUpper(str);
      System.Console.WriteLine(str);
   }
}
```

## Handling errors with exceptions

### The underlying problem: How to properly handle most of the errors which can occur at runtime?

Applications must deal with special situations independently of the program. For example:

- Accessing a file which does not exist.
- Being faced to a memory request when there is insufficient memory available.
- Accessing a server which is not available.
- Accessing a resource without the proper rights.
- The capture of an invalid parameter by the user (such as a birth date in the year 3000).

These situations are not bugs but we can call them errors, which can stop the execution of the program, unless we handle them. To handle such errors, we can test the error codes returned by methods but this approach has two drawbacks:

- The code becomes complicated as every call to a method must be followed by several tests. The tests aren’t centralized and thus violate the principle of code coherency; an important principle in software design.
- The programmer must envision all possible situations during the conception of the program. She must also define the reaction of the program and how to deal with each error type. She cannot easily refactor several types of errors in the same handling code.

In fact, these drawbacks are significant. A solution had to be found to this problem: *exception* management.

## Introduction to exception management in C#

**C++ › C#:** From the point of view of exception management, nothing major has changed compared to C++. However, we will see that several details have changed because .NET exceptions are completely managed by the CLR and not by the operating system.

**C#:** Here are the steps to error management:

1. An error happens;
2. The CLR, the code of the .NET framework or custom code constructs an object which contains some data that describe the error. The details of the implementation of such an object is described a little later;
3. The exception is raised with the object as its parameter;
4. Two possibilities can now occur:
   a) An exception manager catches the exception. It analyzes it and determines the possibility of executing code, for example to save the data or warning the user.
   b) No exception manager catches the exception. The execution of the program ends.

Here is an example where an exception of type `System.DivideByZeroException` is raised: (note that the division by zero of floating-point numbers does not raise an exception).

*Example 14-6*

```
public class Program {
   public static void Main() {
      int i = 1;
      int j = 0;
      int k = i/j;
   }
}
```

The program stops since the exception is not captured and the window in Figure 14-1 is displayed (whether the assembly has been compiled in Debug or Release mode), which offers you to debug your program:

*Figure 14-1: Consequence of an uncaught exception*

Once you are in the Debug mode, *Visual Studio 2005* pops up a very useful assistant to help you visualize the data relating to the exception:

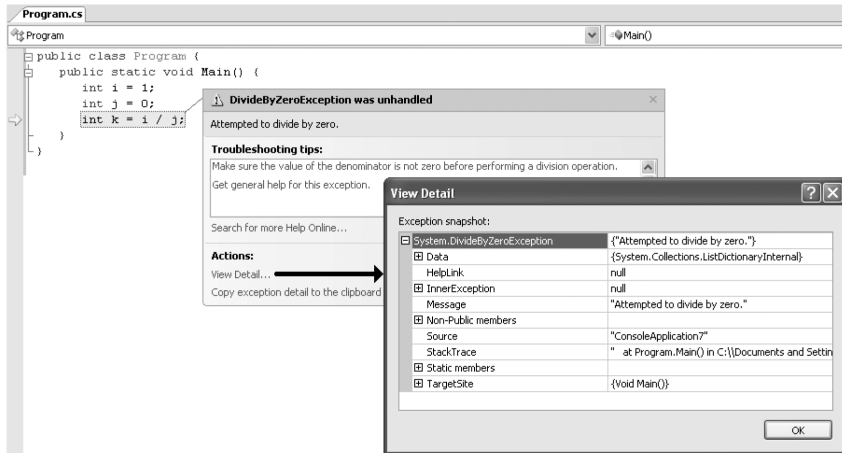


*Figure 14-2: The exception assistant of Visual Studio 2005*

Here is the same example where the exception is caught by an exception manager:

*Example 14-7*

```
using System;
public class Program {
   public static void Main(){
      try {
         int i = 1;
         int j = 0;
         int k = i/j;
      } catch( System.DivideByZeroException ) {
         Console.WriteLine("A division by zero occured!");
      }
   }
}
```

The exception being caught, the program does not stop and displays the following on the console:

```
A division by zero occured!
```

The syntax to define an exception management block is done with two keywords: `try`, `catch`. This syntax is the same as in other languages such as C++, ADA or Java. Let us mention that you can nest several `try/catch` blocks.

## Exception objects and defining custom exception classes

C++ › C#: As with C++, in C#, an exception is represented by an object. The exception can be raised using the `throw new objet` syntax, which also exists in C++. In C++, we had the option of raising an exception without creating an object. In C#, it is required to allocate a new object for each exception explicitly raised using `throw` (except if the exception is raised in a `catch` block, where the current exception can be rethrown).

In C#, this object must be an instance of a class derived from `System.Exception`. In C++, this object can be of any type.

In addition, being a C# object, it must be allocated dynamically (which is not the case in C++).

C#: An exception is always represented by an object. This object generally contains information relative to the problem which caused the exception. This object must be an instance of a class derived from the `System.Exception` class. There are two types of classes which derive from `System.Exception`:

- Those provided by the framework which are thrown by the system but which can also be thrown by your own methods (for example, `System.DivideByZeroException` as seen previously).
- Those that you define yourself and which can only be raised from your own code.

## The System.Exception class

The `System.Exception` class contains properties that you can use from custom exception classes:

- `public string Message{get;}`
  This string contains a descriptive message for the exception. This property is accessible in read only mode but can be initialized by calling the `System.Exception` constructor which accepts a string as a parameter (the property then takes the value of this parameter).
- `public string Source{get;set;}`
  This string contains the name of the object or of the application which generated the error.
- `public string HelpLink{get;set;}`
  This string contains a reference to a web page which explains the exception. You can use this if you put information regarding your own exceptions online.
- `public Exception InnerException{get;}`
  This property references an exception. It is used when a caught exception causes a new exception to be raised. The new exception references the caught exception through this property.

It is recommended to use these properties but it is not a requirement. For example, in the case of exceptions which instantiate the `System.DivideByZeroException` class:

- The `Message` property contains the "`Attempted to divide by zero.`" string.
- The `Source` property is a string equal to the name of the assembly where the exception was raised.
- The `HelpLink` property is an empty string.

Other interesting properties which are automatically initialized by the CLR can be found in this class:

- `public string StackTrace{get;}`
  Contains a representation of the call stack at the moment where the exception was raised (the most recent method being first). It is possible that this string does not contain what should logically be there because some optimizations made by the compiler sometimes modify the structure of the code. If a program is compiled in Debug mode, this string also contains the line number and file name of the instruction which raised the exception.
- `public MethodBase TargetSite{get;}`
  Returns a reference to the `MethodBase` object which references the method that raised the exception.

## Defining custom exception classes

As we have mentioned, you can create your own exception using classes which derive from `System.Exception`. Actually, it is rather recommended to derive custom exception classes from `System.ApplicationException` (which itself derives from the `Exception` class). Here, for example, the definition of an exception class which could be used when an integer argument to a method is outside of a certain range.

*Example 14-8*

```
using System;
public class IntegerOutOfRangeException : ApplicationException {
   public IntegerOutOfRangeException ( int argVal, int inf, int sup ):
      base( string.Format(
                     "The argument value {0} is out of range [{1},{2}]",
                      argVal, inf, sup ) ) { }
}
```

We could have also saved the three integer values in three fields of the class.

## Throwing exception from your code

You have the option of raising an exception (custom or standard) using the `throw` C# keyword. It is necessary to represent this exception using an object, unless the exception is thrown in a `catch` block in which case the current exception is rethrown. Here is an example of the use of `throw` with our own exception class:

*Example 14-9*

```
using System;
public class IntegerOutOfRangeException : ApplicationException {
   public IntegerOutOfRangeException ( int argVal, int inf, int sup ):
      base( string.Format(
                      "The argument value {0} is out of range [{1},{2}]",
                       argVal, inf, sup ) ) { }
}
class Program {
   static void f( int i ){
      // Assume that 'i' must be between 10 and 50 included.
      if( i < 10 || i > 50 )
         throw new IntegerOutOfRangeException ( i , 10 , 50 );
      // Here, we can assert that 'i' is in the range [10;50].
   }
   public static void Main() {
      try {
         f(60);
      }
      catch( IntegerOutOfRangeException e ){
         Console.WriteLine( "Exception: " + e.Message );
         Console.WriteLine(
            "State of the stack when the exception has been raised:"
            + e.StackTrace);
      }
   }
}
```

The execution of this program, when compiled in Debug mode, displays the following:

```
Exception: The argument value 60 is out of range [10,50]
State of the stack when the exception has been raised:
  at Program.f(Int32 i) in d:\my documents\visual studio projects
   \test_exception\program.cs:line 11
  at Program.Main() in d:\my documents\visual studio projects
   \test_exception\program.cs:line 17
```

The execution of this same program, when run in Release mode, displays the following:

```
Exception: The argument value 60 is out of range [10,50]
State of the stack when the exception has been raised:
  at Program.f(Int32 i)
  at Program.Main()
```

You can also raise exceptions defined by the .NET framework. For example, we could have also used the `System.ArgumentOutOfRangeException` class as follows:

*Example 14-10*

```
public class Program {
   static void f(int i) {
      if( i < 10 || i > 50 )
         throw new System.ArgumentOutOfRangeException("i");
   }
   public static void Main() {
      try {
         f(60);
      }
      catch( System.ArgumentOutOfRangeException e ) {
         System.Console.WriteLine( "Exception: " + e.Message );
      }
   }
}
```

This program displays:

```
Exception: Specified argument was out of the range of valid values.
Parameter name: i
```

### No checked exception in C#

Developers who know the Java language may be surprised by the lack of *checked exceptions* in C#. In the article at the following URL `http://www.artima.com/intv/handcuffsP.html` *Anders Hejlsberg* one of the main designers of C#, explains two potential problems caused by controlled exceptions. These happen when a large number of APIs are called and when we must maintain a new version of the application. With the absence of a solution to the problems, the C# language designers preferred not to provide such a mechanism.

## Catch and finally blocks

**C++ › C# :** Exception handlers in the C# language are relatively similar to those of C++ with a few exceptions.

In C#, we do not use the `catch(...)` syntax to catch all exceptions but the `catch(Exception e)` syntax. The advantage is that we still have access to the exception.

The `finally` clause of C# does not exist in C++. It replaces the fact that in C++, we often release the critical resources in the destructors of statically allocated objects. This is not possible in C# as we cannot code the destructor of value types. Another problem is that in C#, the moment where a destructor is called is not determined.

### Notes on catch blocks (exception handlers)

**C#:** An exception handler can contain zero, one or several `catch` blocks, and at most one `finally` block.

In the case where you have several `catch` clauses, the type of exceptions must be different for each block. In addition, at runtime the CLR will test if the type of the exception to catch corresponds to the type of exception handled by `catch` blocks, from the first to the last one. At most, one `catch` block will be executed.

Note that an exception thrown using an instance of the `D` class which is derived from `B`, matches both the `catch(D d)` and the `catch(B b)` clause. The consequences of this are as follows:

- In the same exception handler, the compiler will disallow `catch(D d)` blocks where `D` is a subclass of `B`, after the definition of a `catch(B b)` block. Indeed, such `catch` blocks would not have any chance of being executed.
- The `catch(System.Exception e)` block catches all exceptions since all exception classes are derived from the `System.Exception` class.
- You have the option of having a non-parameterized `catch` block. You simply need to write the `catch` keyword directly followed by the opening brace of the `catch` block. Such a block is equivalent to `catch(System.Exception)`.
- If an exception handler has a non-parameterized `catch` block, it must be the last block. The same applies to `catch(System.Exception)`. If those two blocks are present, the empty `catch` clause must be last.

## *finally block*

If a `finally` block is present, it must be located after all `catch` blocks. The `finally` block of code is executed in all possible cases which are:

- No exception was raised in the `try` block.
- An exception was raised in the `try` block and was caught by a `catch` block.
- An exception was raised in a `try` clause and was caught by a `catch` block. An exception is then raised in this block.
- An exception was raised in the `try` block and was not caught by a `catch` block.

A `finally` clause is generally used to free critical resources (such as a database connection or an opened file) independently of whether an exception was thrown or not. Note that if an exception is raised in a `finally` block (which is not advisable) while a raised exception could not be caught by the current exception handler, this new exception replaces the previous one. For example:

*Example 14-11*

```
using System;
public class Program {
   public static void Main(){
      try {
         try {
            throw new ArgumentOutOfRangeException();
         }
         catch( DivideByZeroException e ) {
            Console.WriteLine( “Exception handler 1:” );
            Console.WriteLine( “Exception: “ + e.Message );
         }
         finally {
            Console.WriteLine( “finally 1” );
            throw new DivideByZeroException();
         }
      }
      catch(Exception e) {
```

*Example 14-11*

```
         Console.WriteLine( "Exception handler 2:" );
         Console.WriteLine( "Exception: " + e.Message );
      }
      finally {
         Console.WriteLine( "finally 2" );
      }
   }
}
```

This program displays:

```
finally 1
Exception handler 2:
Exception: Attempted to divide by zero.
finally 2
```

It is often preferable to use a `finally` clause indirectly through the use of the *Dispose pattern* described at page 351.

### Increasing exception semantics

When an exception bubbles up the different embedded calls, it has a tendency to loose signification as it traverses several code layers. It is then often necessary to catch the exception to throw a new one, where the content depends on the caught exception. To not loose anything of the initial exception, you can reference it directly in the new exception using the `InnerException` property of the `Sytem.Exception` class.

## Exceptions thrown from a constructor or from a finalizer

C++ › C#: In C#, an exception thrown from a constructor is treated exactly like any other exceptions, and the object is not created. We will interest ourselves to the behavior of the CLR when an exception is thrown from a class constructor.

In C#, an exception raised from the `Finalize()` method and which is not caught in the constructor does not cause any major problems.

### Exception thrown from an instance constructor

C#: An exception thrown from an instance constructor is treated exactly as any other exception and the object is not created. It is however recommended not to raise an exception from a constructor. Here is a program which illustrates this:

*Example 14-12*

```
using System;
public class Article {
   public Article() { i=3; throw new ArgumentOutOfRangeException(); }
   public Article( int j ) { i=j; }
   public int i=0;
}
public class Program {
   public static void Main() {
      Article article = new Article( 2 );
      try{
```

*Example 14-12*

```
         article = new Article();
      } catch( Exception e ) {
         Console.WriteLine("Exception: "+e.Message);
      }
      Console.WriteLine( article.i );
   }
}
```

This program displays:

```
Exception: Specified argument was out of the range of valid values.
2
```

Understand that allocation of the second instance of `Article`, in the `try` block, has failed from the fact that an exception was thrown from the constructor without being caught.

## *Exception thrown from a class constructor or while initializing a static field*

Calls to class constructors and the initialization of static fields are non-deterministic, meaning that the rules which define when they will be called depend on the implementation of the CLR and are not documented. More details on this topic can be found at page 357. However, an exception can be raised in one of these methods and not be caught. In this case, all happens as if an exception of type `TypeInitializationException` is raised at the place (which is not determined) in the program which triggered the call to the static constructor. In general, this location is the first instantiation of the class or where the first access to one of its static members occurs. The exception which was initially raised in the class constructor is then referenced by the `Exception.InnerException` field of the `TypeInitializationException` exception. This behavior is automatically managed by the CLR. Here is an example which illustrates this:

*Example 14-13*

```
using System;
public class Article {
   static  Article() {
      throw new ArgumentOutOfRangeException();
   }
}
public class Program {
   public static void Main() {
      try{
         Article article = new Article();
      }
      catch(Exception e) {
         Console.WriteLine("Exception: " + e.Message );
         Console.WriteLine("Inner Exception: "+ e.InnerException.Message );
      }
   }
}
```

This program displays the following:

```
Exception: The type initializer for "Article" threw an exception.
Inner Exception: Specified argument was out of the range of valid values.
```

### Exceptions thrown from a finalizer

An uncaught exception thrown from a finalizer has for effect of immediately exiting this method within the thread dedicated to the execution of finalizers. However, the memory allocated for the object is still released by the garbage collector.

If the exception was thrown from a finalizer of a subclass, the finalizer of the base class is still executed.

It is strongly recommended not to throw exceptions from finalizers.

## Exception handling and the CLR

You must understand that the CLR completely takes care of the management of .NET exceptions. It has the following responsibilities:

- The CLR has the responsibility of finding the proper the proper `catch()` block by traversing the call stack. The CLR stores information about this search in the object representing the exception. Amongst this we find the call stack of the method which leads to the exception. Hence the developer can sift through this information in order to understand the cause of the exception.
- When the exception is raised in a distributed environment, the CLR takes care of serializing and propagating the exception to the AppDomain containing the client. Be aware that there is however certain precautions to take if you wish custom exceptions to transit through .NET Remoting (more information on this topic can be found at `http://www.thinktecture.com/Resources/RemotingFAQ/CustomExceptions.html`).
- The CLR sometimes has the responsibility of catching an exception to rethrow another which has more meaning. For example, we have seen in the previous section that the CLR propagates a `TypeInitializationException` exception no matter what the type of the exception thrown from a class constructor. This is also the case for exception thrown from methods called using late binding invocation. Note that in this type of case, the exception initially thrown is stored in the `InnerException` property of the exception.
- At page 236 we explain that the CLR transforms `HRESULT` error codes returned by COM objects into managed exception. Also, when a managed object is considered as a COM object, the CLR produces a `HRESULT` from a managed exception which goes back into unmanaged code. We will see in the next section how the CLR deals with *Windows* native exceptions.
- At page 75 we explain that the `AppDomain` class presents the `UnhandledException` event. The following example shows how this event can be used to execute code when the CLR notices that the current process will crash because an exception was not caught. We explain in the comments the operations which are generally needed before ending the process:

*Example 14-14*

```
using System;
using System.Threading;
public class Program {
   public static void Main() {
      Console.WriteLine("Thread{0}: Hello world...",
                        Thread.CurrentThread.ManagedThreadId );
      AppDomain currentDomain = AppDomain.CurrentDomain;
      currentDomain.UnhandledException += UnhandledExceptionHandler;
      throw new Exception ("The exception explanation goes here.");
   }
   static void UnhandledExceptionHandler(
                 object s, UnhandledExceptionEventArgs e) {
```

*Example 14-14*

```
      Console.WriteLine("Thread{0}: UnhandledExceptionHandler: {1}",
                        Thread.CurrentThread.ManagedThreadId ,
                       (e.ExceptionObject as Exception).Message);
      //  a) Save an error report.
      //  b) Ask the user if he wishes to save the current state.
      //  c) Ask the user if he wishes that the error report is
      //     automatically sent to the development team.
   }
}
```

The program displays this before crashing:

```
Thread1: Hello world...
Thread1: UnhandledExceptionHandler: The exception explanation goes here.
```

- At page 102 we explain advanced features specific to the CLR which allow to improve the reliability of an application which may be faced to asynchronous exceptions.

## Unmanaged exceptions

Typically, the underlying *Windows* thread to a managed thread traverses managed code (which is compiled in native code) and some native code. When a managed exception happens within a block of managed code, the CLR makes sure to find the proper `catch` clause in the managed code. Regions of native code which are eventually located between the raised exception and the managed `catch` clause are unstacked normally.

*Windows* offers a mechanism named *Structured Exception Handling* (*SEH*) to manage native exceptions. A detailed description of this mechanism can be found in an article written by *Matt Pietrek* `http://www.microsoft.com/msj/0197/exception/exception.aspx`.

The CLR is capable of detecting when a native exception goes up into managed code. In this situation, it yields a managed exception which depends on the type of native exception intercepted:

| Native exception code | Managed exception type |
|---|---|
| STATUS_FLOAT_INEXACT_RESULT<br>STATUS_FLOAT_INVALID_OPERATION<br>STATUS_FLOAT_STACK_CHECK<br>STATUS_FLOAT_UNDERFLOW | System.ArithmeticException |
| STATUS_FLOAT_OVERFLOW<br>STATUS_INTEGER_OVERFLOW | System.OverflowException |
| STATUS_FLOAT_DIVIDE_BY_ZERO<br>STATUS_INTEGER_DIVIDE_BY_ZERO | System.DivideByZeroException |
| STATUS_FLOAT_DENORMAL_OPERAND | System.FormatException |
| STATUS_ACCESS_VIOLATION | System.NullReferenceException |
| STATUS_ARRAY_BOUNDS_EXCEEDED | System.IndexOutOfRangeException |
| STATUS_NO_MEMORY | System.OutOfMemoryException |
| STATUS_STACK_OVERFLOW | System.StackOverflowException |
| All other codes | System.Runtime.InteropServices.SEHException |

The SEH mechanism works with an exception filter model which are registered to the thread. These filters allow telling *Windows* the native function to call when a native exception is detected on a thread. The CLR registers its own filter on each thread which executes managed code. The registered native function has the responsibility of triggering the `AppDomain.UnhandledException`

event. If a thread was not created by the CLR (i.e. it was not created using a call to the `Thread.Start()` method) there is always a risk that exception filters registered on this thread by other component may interfere with the filters registered by the CLR. Also, be aware that in this particular case, the `UnhandledException` may not necessarily be triggered.

## Exception handling and Visual Studio

When the *Visual Studio* environment debugs an application, it can suspend execution by returning control to the debugger when an exception (managed or not) is raised or when it is not caught. This feature can be configured by the type of exception through the *Debug Exceptions…* menu. The list of exceptions is divided into five categories:

- **C++ Exceptions**: lists the unmanaged C++ exceptions.
- **Common Language Runtime Exceptions**: lists the managed exception (categorized by their namespace).
- **Managed Debugging Assistants**: lists problematic events known by the CLR which can sometimes result into a managed exception. Hence, if you wish to confirm that such an exception is the consequence of a certain event or if you wish to be informed of such an event when it does not result into an exception, you must use this list. These events are described on **MSDN** in the article named **Diagnosing Errors with Managed Debugging Assistants**.
- **Native Run-Time Checks**: lists critical exceptions which can occur in a C/C++ program
- **Win32 Exceptions**: lists the SHE exception codes.

*Visual Studio* also allows you to add custom exception types to this list.

## Guidelines on exception management

### When should you consider throwing an exception?

The exception mechanism is generally well understood but quite often used improperly. **The base principle is that an application which functions in a normal way should not throw exceptions.** This forces us to define what an abnormal situation is. There are three types:

- Those which happen because of a problem with the execution environment but can be solved by a modification to this environment (missing file, invalid password, non-well-formed XML document, network unavailability, restricted security permissions…). Here we talk of *business exception*.
- Those which happen because of an execution environment problem which cannot be solved. For example memory hungry applications such a *SQL Server 2005* may be limited to 2 or 3GB of addressing space in a 32-bits *Windows* process. Here we talk of *asynchronous exceptions* from the fact that they are not related to the semantic of the code which raised it. To manage this type of problem, you must use advanced CLR features described at page 102. This is essentially equivalent to treating such abnormal situation as normal! **Be aware that only the large servers with push the limits of its resources should encounter asynchronous exceptions and will need to use these mechanisms.**
- Those which happen because of a bug and which can only be solved by a new version which fixes properly the bug.

### What to do in exception handlers?

When you catch an exception, you can envision three scenarios:

- Either you are faced with a real problem but that you can address it by fixing the conditions which cause the problem. For this, you may need new information (invalid password › ask the user to reenter the password…).
- Either you are faced with a problem which you cannot resolve at this level. In this case, the only good approach is to rethrow the exception. It is possible that there may not be a proper exception handler and in this case, you delegate the decision to the runtime host. In console or windowed applications, the runtime host causes the whole process to terminate. Note that you can use the `AppDomain.UnhandledException` event which is triggered in this situation in order to take over the termination of the process. You can take advantage of this 'last chance' to save your data (as with *Word*) which without this would definitely lead to data loss. In an ASP.NET context, an error processing mechanism, described at page 750 is put in place.
- In theory, a third scenario can be envisioned. It is possible that the exception that was caught represents a false alarm. In practice, this never happens.

You must not catch an exception to simply log it and then rethrow it. To log exceptions and the code that they have traversed, we recommend using less intrusive approaches such as the use of specialized events of the `AppDomain` class or the analysis of the methods on the stack at the moment where the exception was thrown.

You must not release the resources that you have allocated when you catch an exception. Also be aware that in general only unmanaged resources are susceptible of causing problems (such as memory leaks). This type of code to release resources must be placed in the `finally` block or in a `Dispose()` method. In C#, the `finally` blocks are often implicitly encapsulated in a `using` block which acts on objects implementing the `IDisposable` interface.

## *Where should you put exception handlers?*

For a specific type of exception, asking this question comes down to asking yourself at which method depth this exception must be caught and what must be done about it. By method depth, we means the number of calls embedded since the entry point (generally the `Main()` method). This means that the method representing the entry point is the least deep. The answer to these two questions depends on the semantics of an exception. Ask yourself for each type of exception, at which depth your code is more apt to be able to correct the conditions which have triggered the exception and resume the execution or to be able to properly terminate the application.

Generally, the deeper a method is, the less it must catch custom exceptions. The reason is that custom exceptions often have a signification to the business of your application. Hence, if you develop a class library, you must let exceptions which are meant to the client application bubble outside of the library.

## *Exceptions vs. returned error code*

You may be tempted to use exceptions instead of returning error codes in your methods, in order to indicate a potential problem. You must be careful as the use of exceptions suffers from two major disadvantages:

- The code is hard to read. In fact, to understand the code, you must manually do the work of the CLR which consists in traversing the calls until you find an exception handler. Even if you properly separate your calls into layers, the code is still difficult to read.
- Exception handling by the CLR is much more expensive than simply looking at an error code.

The fundamental rule mentioned at the beginning of this section can help you make this decision: **an application which functions within normal conditions does not raise exceptions**.

### Never under estimate bugs whose consequences are caught by exception handlers

An abusive use of exceptions happen when we assume that, since we catch all exceptions, those provoked by eventual bugs will also be caught. We then assume that they will prevent the application from crashing. This reasoning does not take into account the fact that the main nuisances from bugs are those which goes uncaught such as indeterminist, unexpected or false results.

## Anonymous methods

The current section as well as the following section presents two new features added to the 2.0 version of the C# language. Unlike generics, these two features do not involve new IL instructions. All the magic happens at the level of the compiler.

Each of these two sections will have the same structure: a 'classic' presentation of the feature followed by a more detailed explanation of the work done by the compiler in order to explain how you can make full use of the feature.

### Introduction to C#2 anonymous methods

Let's begin by enhancing some C#1 code to use C#2 anonymous methods . Here is a simple C# program that first references and then invokes a method, through a delegate:

*Example 14-15*

```
class Program {
   delegate void DelegateType();
   static DelegateType GetMethod(){
      return new DelegateType(MethodBody);
   }
   static void MethodBody() {
      System.Console.WriteLine("Hello");
   }
   static void Main() {
      DelegateType delegateInstance = GetMethod();
      delegateInstance();
      delegateInstance();
   }
}
```

Here is the same program rewritten using a C#2 anonymous method:

*Example 14-16*

```
class Program {
   delegate void DelegateType();
   static DelegateType GetMethod() {
      return delegate() { System.Console.WriteLine("Hello"); };
   }
   static void Main() {
      DelegateType delegateInstance = GetMethod();
      delegateInstance();
      delegateInstance();
   }
}
```

You should notice that:

- The `delegate` keyword has a new use in C#2. When the C#2 compiler finds the delegate keyword inside the body of a, it expects it to be followed by an anonymous method body.
- It is possible to assign an anonymous method to a delegate reference
- We understand why this feature is named anonymous method: the method defined in the body of `GetMethod()` is not named. Nevertheless, it is possible to invoke it because it is referenced by a delegate instance.

You should notice as well that it is possible to use the operator += to allow a delegate instance to reference several methods (anonymous or not):

*Example 14-17*

```
using System;
class Program{
   delegate void DelegateType();
   static void Main(){
      DelegateType delegateInstance = delegate() {
         Console.WriteLine("Hello"); };
      delegateInstance += delegate() { Console.WriteLine("Bonjour"); };
      delegateInstance();
   }
}
```

As you might expect this program outputs:

```
Hello
Bonjour
```

## *Anonymous methods can accept arguments*

As shown in the following example, an anonymous method can accept arguments of any type. You can also use keywords `ref` and `out` to tune how arguments are passed to the method:

*Example 14-18*

```
class Program {
   delegate int DelegateType( int valTypeParam, string refTypeParam,
                              ref int refParam, out int outParam);
   static DelegateType GetMethod() {
      return delegate( int valTypeParam , string refTypeParam,
                       ref int refParam , out int outParam     ) {
         System.Console.WriteLine( "Hello valParam:{0} refTypeParam:{1}",
                                   valTypeParam, refTypeParam);
         refParam++;
         outParam = 9;
         return valTypeParam;
      }; // End of the body of the anonymous method.
   }
   static void Main() {
      DelegateType delegateInstance = GetMethod();
      int refVar = 5;
      int outVar;
      int i = delegateInstance( 1, "one", ref refVar, out outVar );
      int j = delegateInstance( 2, "two", ref refVar, out outVar );
      System.Console.WriteLine( "i:{0} j:{1} refVar:{2} outVar:{3}",
                                i, j, refVar, outVar);
   }
}
```

This program outputs:

```
Hello valParam:1 refTypeParam:one
Hello valParam:2 refTypeParam:two
i:1 j:2 refVar:7 outVar:9
```

As you can see, the returned type is not defined inside the anonymous method declaration. The returned type of an anonymous method is inferred by the C# v2 compiler from the returned type of the delegate to which it is assigned. This type is always known because the compiler forces the assignment of any anonymous method to a delegate.

An anonymous method can't be tagged with an attribute. This restriction implies that you can't use the `param` keyword in the list of arguments of an anonymous method. Indeed, using the keyword `param` forces the compiler to tag the concerned method with the `ParamArray` attribute.

*Example 14-19*

```
using System;
class Program {
   delegate void DelegateType( params int[] arr );
   static DelegateType GetMethod() {
      // Compilation error: param is not valid in this context.
      return delegate( params int[] arr ){ Console.WriteLine("Hello");};
   }
}
```

## A syntax subtlety

It is possible to declare an anonymous method without any signature, i.e. you are not compelled to write a pair of parenthesis after the keyword `delegate` if your anonymous method doesn't take any argument. In this case, your method can be assigned to any delegate instance that returns a void type and that doesn't have an out arguments. Obviously, such an anonymous method doesn't have access to the parameters that are provided through its delegate invocation.

*Example 14-20*

```
using System;
class Program{
   delegate void DelegateType(int valTypeParam, string refTypeParam,
                              ref int refParam);
   static void Main() {
      DelegateType delegateInstance = delegate {
         Console.WriteLine( "Hello" ); };
      int refVar = 5;
      delegateInstance( 1, "one", ref refVar );
      delegateInstance( 2, "two", ref refVar );
   }
}
```

## Anonymous methods and generics

As shown in the example below, an argument of an anonymous method can be of a generic type:

*Example 14-21*

```
class Foo<T> {
   delegate void DelegateType( T t );
   internal void Fct( T t ) {
      DelegateType delegateInstance = delegate( T arg ){
```

*Example 14-21*

```
         System.Console.WriteLine( "Hello arg:{0}" , arg.ToString() ); };
      delegateInstance( t );
   }
}
class Program {
   static void Main() {
      Foo<double> inst = new Foo <double>();
      inst.Fct(5.5);
   }
}
```

In .NET 2, a delegate type can be declared with generic arguments. An anonymous method can be assigned to a delegate instance of such a type. You just have to resolve type parameters on both side of the assignment:

*Example 14-22*

```
class Program{
   delegate void DelegateType<T>( T t );
   static void Main() {
      DelegateType<double> delegateInstance = delegate( double arg ) {
         System.Console.WriteLine( "Hello arg:{0}" , arg.ToString() );
      };
      delegateInstance(5.5);
   }
}
```

## *Use of anonymous methods in the real world*

Anonymous methods are particularly suited to define 'small' methods that must be invoked through a delegate. For instance, you might use an anonymous method to code the entry point procedure of a thread:

*Example 14-23*

```
using System.Threading;
class Program{
   static void Main(){
      Thread thread = new Thread( delegate() {
        System.Console.WriteLine( "ManagedThreadId:{0} Hello",
                                  Thread.CurrentThread.ManagedThreadId );
      } );
      thread.Start();
      System.Console.WriteLine( "ManagedThreadId:{0} Bonjour",
                                Thread.CurrentThread.ManagedThreadId );
   }
}
```

This program displays:

```
ManagedThreadId:1 Bonjour
ManagedThreadId:3 Hello
```

Another classic example of this kind of use lies in the *Windows Forms* control event callbacks:

*Example 14-24*

```
public class FooForm : System.Windows.Forms.Form {
   System.Windows.Forms.Button m_Button;
   public FooForm() {
      InitializeComponent();
      m_Button.Click += delegate( object sender, System.EventArgs args ) {
         System.Windows.Forms.MessageBox.Show("m_Button Clicked");
      };
   }
   void InitializeComponent()  {/*...*/}
}
```

It seems that anonymous method looks like a tiny language enhancement. It's now time to dig under the hood to realize that anonymous methods are far more complex and can be far more useful.

## The C#2 compiler and anonymous methods

### The easy way

As you might expect, when an anonymous method is compiled, a new method is created by the compiler in the concerned class:

*Example 14-25*

```
class Program {
   delegate void DelegateType();
   static void Main() {
      DelegateType delegateInstance = delegate() {
         System.Console.WriteLine("Hello"); };
      delegateInstance();
   }
}
```

The following assembly is the compiled version of the previous program (the assembly is viewed using the Reflector tool introduced at page 19):

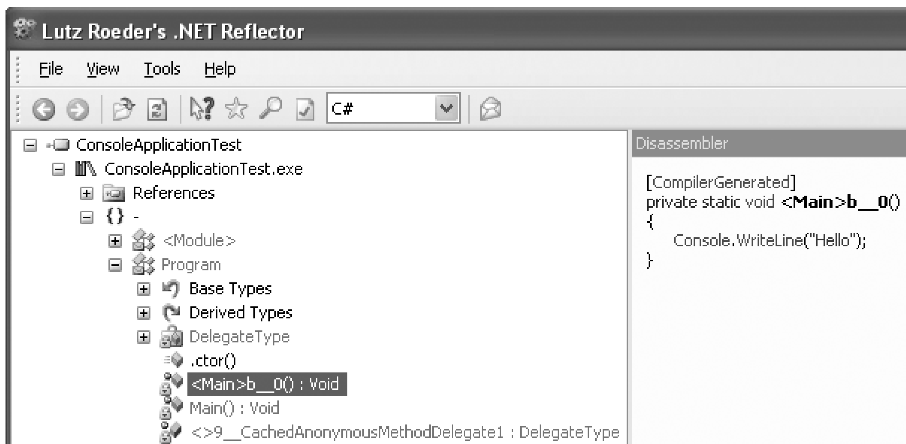


*Figure 14-3: Easy way of anonymous method compilation*

Indeed, a new private and static method named `<Main>b__0()` has been automatically generated and contains the code of our anonymous method. If our anonymous method was declared inside an instance method, the generated method would have been an instance method.

We also note that a delegate field named `<>9_CachedAnonymousMethoddelegate1` of type `delegateType` has been generated to reference our anonymous method.

It is interesting to note that all these generated members can't be viewed with the C# intellisense because their names contain a pair of angle brackets < >. Such names are valid for the CLR syntax but incorrect for the C# syntax.

## *Captured local variable*

To keep things clear and simple, we haven't mentioned yet the fact that an anonymous method can have access to a local variable of its outer method. Let's analyze this feature through the following example:

*Example 14-26*

```
class Program {
   delegate int DelegateTypeCounter();
   static DelegateTypeCounter MakeCounter(){
      int counter = 0;
      DelegateTypeCounter delegateInstanceCounter =
         delegate { return ++counter; };
      return delegateInstanceCounter;
   }
   static void Main() {
      DelegateTypeCounter counter1 = MakeCounter();
      DelegateTypeCounter counter2 = MakeCounter();
      System.Console.WriteLine( counter1() );
      System.Console.WriteLine( counter1() );
      System.Console.WriteLine( counter2() );
      System.Console.WriteLine( counter2() );
   }
}
```

This program outputs:

```
1
2
1
2
```

Think about it, it might stump you. The local variable counter seems to survive when the thread leaves the `MakeCounter()` method. Moreover, it seems that two instances of this 'surviving' local variable exist!

Note that in .NET 2, the CLR and the IL language haven't been tweaked to support the anonymous method feature. The interesting behavior must stem from the compiler. It's a nice example of 'syntactic sugar'. Let's analyze the assembly with the Reflector tool (see Figure 14-4).

This analysis makes things clear because:

- The compiler doesn't only create a new method as we saw in the previous section. It utterly creates a new class named `<>c__DisplayClass1` in this example.
- This class has an instance method called `<MakeCounter>b__0()`. This method has the body of our anonymous method.

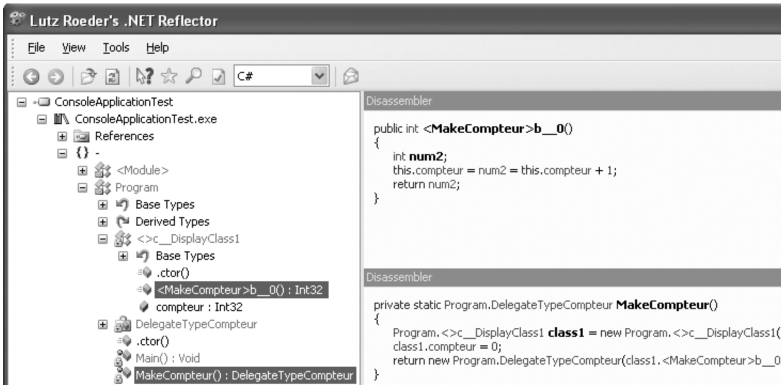


*Figure 14-4: Anonymous method and captured local variable*

- This class has also an instance field called `counter`. This field keeps track of the state of the local variable `counter`. We say the local variable `counter` has been **captured** by the anonymous method.
- The method instantiates the class <>c__DisplayClass1. Moreover it initializes the field counter of the created instance.

Notice that the `MakeCounter()` method doesn't have any local variable. For the `counter` variable, it uses the same field of the generated instance of the class <>c__DisplayClass1.

Before explaining why the compiler has this surprising behavior, let's go further to get a thorough understanding of its work.

## *Captured local variables and code complexity*

The following example is more subtle than expected:

*Example 14-27*

```
using System.Threading;
class Program {
   static void Main() {
      for (int i = 0; i < 5; i++)
         ThreadPool.QueueUserWorkItem( delegate {
            System.Console.WriteLine(i); }, null);
   }
}
```

This program outputs in a non-deterministic way something like:

```
0
1
5
5
5
```

This result compels us to infer that the local variable `i` is shared amongst all threads. The execution is non-deterministic because the `Main()` method and our closure (or anonymous methods) are executed simultaneously by several threads. To make things clear, here is the decompiled code of the `Main()` method:

```
private static void Main(){
   bool flag1;
   Program.<>c__DisplayClass1 class1 = new Program.<>c__DisplayClass1();
   class1.i = 0;
   goto Label_0030;
Label_000F:
   ThreadPool.QueueUserWorkItem(new WaitCallback(class1.<Main>b__0), null);
   class1.i++;
Label_0030:
   flag1 = class1.i < 5;
   if ( flag1 ) {
      goto Label_000F;
   }
}
```

Notice that the fact that the value of 5 being printed indicates that the `Main()` method is done executing the loop when the display is done. The following version of this program has a deterministic execution:

*Example 14-28*

```
using System.Threading;
class Program {
   static void Main() {
      for (int i = 0; i < 5; i++){
         int j = i;
         ThreadPool.QueueUserWorkItem(delegate {
            System.Console.WriteLine(j); }, null);
      }
   }
}
```

This time, the program outputs:

```
0
1
2
3
4
```

This behavior stems from the fact that the local variable `j` is captured for each iteration. Here is the decompiled code of the `Main()` method:

```
private static void Main(){
   Program.<>c__DisplayClass1 class1;
   bool flag1;
   int num1 = 0;
   goto Label_0029;
Label_0004:
   class1 = new Program.<>c__DisplayClass1();
   class1.j = num1;
   ThreadPool.QueueUserWorkItem(new WaitCallback(class1.<Main>b__0), null);
   num1++;
Label_0029:
   flag1 = num1 < 5;
   if (flag1) {
      goto Label_0004;
   }
}
```

This sheds light on the fact that capturing local variables with anonymous methods is not an easy thing. You should always take care when using this feature.

Note that a captured local variable is no longer a local variable. If you access such a variable with some unsafe code, you might have pin it before (with the C# keyword `fixed`).

## *An anonymous method accesses to an argument of the outer method*

Arguments of a method can always be deemed as local variables. Therefore, C#2 allows an anonymous method to use arguments of its outer method. For instance:

*Example 14-29*

```
using System;
class Program {
   delegate void DelegateTypeCounter();
   static DelegateTypeCounter MakeCounter( string counterName ) {
      int counter = 0;
      DelegateTypeCounter delegateInstanceCounter = delegate{
         Console.WriteLine( counterName + (++counter).ToString() );
      };
      return delegateInstanceCounter;
   }
   static void Main() {
      DelegateTypeCounter counterA = MakeCounter("Counter A:");
      DelegateTypeCounter counterB = MakeCounter("Counter B:");
      counterA();
      counterA();
      counterB();
      counterB();
   }
}
```

This program outputs:

```
Counter A:1
Counter A:2
Counter B:1
Counter B:2
```

Nevertheless, an anonymous method can't capture an `out` or `ref` argument. This restriction is easy to understand as soon as you realize that such an argument can't be seen as a local variable. Indeed, such an argument survives the execution of the method.

## *An anonymous method accessing a member of the outer class*

An anonymous method can access members of its outer class. The case of static member access is easy to understand since there is one and only one occurrence of any static field in the domain application. Thus, there is nothing like 'capturing' a static field.

The access to the instance of a member is less obvious. To clarify this point, remember that the `this` reference that allows access to instance members, is a local variable of the outer instance method. Therefore, the `this` reference is captured by the anonymous method. Let's analyze the following example:

*Example 14-30*

```
delegate void DelegateTypeCounter();
class CounterBuilder {
   string m_Name; // An instance field
   internal CounterBuilder( string name ) { m_Name = name; }
   internal DelegateTypeCounter BuildCounter( string counterName ) {
      int counter = 0;
      DelegateTypeCounter delegateInstanceCounter = delegate {
         System.Console.Write( counterName +(++counter).ToString() );
         // we could have written this.m_Name.
         System.Console.WriteLine(" Counter built by: " + m_Name);
      };
      return delegateInstanceCounter;
   }
}
class Program {
   static void Main() {
      CounterBuilder cBuilder1 = new CounterBuilder( "Factory1" );
      CounterBuilder cBuilder2 = new CounterBuilder( "Factory2" );
      DelegateTypeCounter cA = cBuilder1.BuildCounter( "Counter A:" );
      DelegateTypeCounter cB = cBuilder1.BuildCounter( "Counter B:" );
      DelegateTypeCounter cC = cBuilder2.BuildCounter( "Counter C:" );
      cA();  cA ();
      cB();  cB();
      cC();  cC();
   }
}
```

This program outputs:

```
Counter A:1 Counter built by: Factory1
Counter A:2 Counter built by: Factory1
Counter B:1 Counter built by: Factory1
Counter B:2 Counter built by: Factory1
Counter C:1 Counter built by: Factory2
Counter C:2 Counter built by: Factory2
```

Let's decompile the `MakeCounter()` method to expose the capture of the `this` reference:

```
internal DelegateTypeCounter BuildCounter(string counterName){
    CounterBuilder.<>c__DisplayClass1 class1 = new
                                 CounterBuilder.<>c__DisplayClass1();
    class1.<>4__this = this;
    class1.counterName = counterName;
    class1.counter = 0;
    return new DelegateTypeCounter(class1.<BuildCounter>b__0);
}
```

Notice that the `this` reference cannot be captured by an anonymous method that is defined in a structure. Here is the compiler error:

Anonymous methods inside structs cannot access instance member of 'this'. Consider copying 'this' to a local variable outside the anonymous method and using the local instead.

# Advanced uses of anonymous methods

## Definitions: closure and lexical environment

A *closure* is a function that captures values of its lexical environment, **when** it is created at run-time. The *lexical environment* of a function is the set of variables visible **from** the concerned function.

In previous definitions, we carefully used the terms **when** and **from**. It indicates that the notion of closure pinpoints something that exists at run-time (as the concept of object). It also indicates that the notion of lexical environment pinpoints to something that exists in the code, i.e. at compile-time (as the concept of class). Consequently, you can consider that the lexical environment of a C#2 anonymous method is the class generated by the compiler. Following the same idea, you can consider that an instance of such a generated class is a closure.

The definition of a closure also implies the notion of creating a function at run-time. Mainstream *imperative languages* such as C, C++, C#1, Java or VB.NET1 don't support the ability to create an instance of a function at run-time. This feature stems from *functional languages* such as Haskell or Lisp. Thus C#2 goes beyond imperative languages by supporting closures. However, C#2 is not the first imperative language that supports closures since Perl and Ruby also have this feature.

C# › C++: The concept of closure is close to the concept of *functor* described at page 492.

## Ramblings on closures

A function computes its results both from values of its arguments and from the context that surrounds its invocation. You can consider this context as a set of background data. Thus, arguments of a function can be seen as foreground data. Therefore, the decision that an input data of a function must be an argument must be taken from the relevance of the argument for the computation.

Generally, when using object languages, the context of a function (i.e. the context of an instance method) is the state of the object on which it is invoked. When programming with non object oriented imperative languages such as C, the context of a function is the values of global variables. When dealing with closures, the context is the values of captured variables when the closure is created. Therefore, as classes, closures are a way to associate behavior and data. In object oriented world, methods and data are associated thanks to the `this` reference. In functional world a function is associated with the values of captured variables. To make thinks clear:

- You can think of an object as a set of method attached to a set of data.
- You can think of a closure as a set of data attached to a function.

## Using closures instead of classes

The previous section implies that some type of classes could be replaced by some anonymous methods. Actually, we already perform such replacement in our implementation of counter. The behavior is the increment of the counter while the state is its value. However, the counter implementation doesn't harness the possibility of passing arguments to an anonymous method. The following example shows how to harness closures to perform parameterized computation on the state of an object:

*Example 14-31*

```
class Program {
   delegate void DelegateMultiplier( ref int integerToMultipl);
   static DelegateMultiplier BuildMultiplier ( int multiplierParam ) {
      return delegate( ref int integerToMultiply ) {
         integerToMultiply *= multiplierParam;
      };
   }
   static void Main() {
      DelegateMultiplier multiplierBy8 = BuildMultiplier(8);
      DelegateMultiplier multiplierBy2 = BuildMultiplier(2);
      int anInteger = 3;
      multiplierBy8( ref anInteger );
      // Here, anInteger is equal to 24.
      multiplierBy2( ref anInteger );
      // Here, anInteger is equal to 48.
   }
}
```

Here is another example that shows how to harness closures to perform parameterized computation in order to obtain a value from the state of an object:

*Example 14-32*

```
using System;
class Article {
   public Article( decimal price ) { m_Price = price; }
   private decimal m_Price;
   public decimal Price { get { return m_Price; } }
}
class Program {
   delegate decimal DelegateTaxComputer( Article article );
   static DelegateTaxComputer BuildTaxComputer( decimal tax ) {
      return delegate( Article article ) {
         return ( article.Price * (100 + tax) ) / 100;
      };
   }
   static void Main(){
      DelegateTaxComputer taxComputer19_6 = BuildTaxComputer(19.6m);
      DelegateTaxComputer taxComputer5_5 = BuildTaxComputer(5.5m);
      Article article = new Article(97);
      Console.WriteLine("Price TAX 19.6% : " + taxComputer19_6(article) );
      Console.WriteLine("Price TAX  5.5% : " + taxComputer5_5(article) );
   }
}
```

Understand that all the power behind the use of closures in both previous examples comes from the fact that they prevent us from creating small classes (which are in fact created implicitly by the compiler).

## Delegates and closures

By taking a closer look, we notice that the notion of a delegate used on an instance method in .NET 1.x is conceptually close to the notion of closure. In fact, such a delegate references both data (the state of the object) and a behavior. A constraint does exist: the behavior must be an instance method of the class defining the type of the `this` reference.

This constraint is minimized in .NET 2. Because of certain overloads of the `Delegate.CreateDelegate()` method, you can now reference the first argument of a static method in a delegate. For example:

*Example 14-33*

```
class Program {
   delegate void DelegateType( int writeNTime );
   // This method is public to avoid problems of reflection
   // on a non-public member.
   public static void WriteLineNTimes( string s, int nTime ) {
      for( int i=0; i<nTime; i++ )
         System.Console.WriteLine( s );
   }
   static void Main() {
      DelegateType deleg = System.Delegate.CreateDelegate(
         typeof( DelegateType ),
         "Hello",
         typeof(Program).GetMethod( "WriteLineNTimes" )) as DelegateType;
      deleg(4);
   }
}
```

This program displays:

```
Hello
Hello
Hello
Hello
```

Finally note that internally, the implementation of delegates has been completely revised in the 2.0 version of the framework and the CLR. The good news is that the invocation of a method through a delegate is now much more efficient.

### Using anonymous methods to handle collections

Aat page 493 we explain how to use the syntax of anonymous methods to greatly improve the syntax needed to manipulate collections.

## C#1 iterators

### Enumerables, enumerators and the iterator design pattern

It is necessary to get a good understanding on what enumerables and enumerators are before covering C#1 and C#2 enumerators. An object is an *enumerable* if it contains or references a collection of objects and if elements of this collection can be enumerated with the `foreach` C# keyword. Instances of classes that implement `System.Collections.IEnumerable` are enumerables. As we'll see, there exists some enumerables that don't fulfill this condition. An object is an *enumerator* if its class implements the interface `System.Collections.IEnumerator`. Here are the definitions of these interfaces:

```
public interface System.Collections.IEnumerable {
   System.Collections.IEnumerator GetEnumerator();
}
public interface System.Collections.IEnumerator {
   object Current { get; }
   bool MoveNext();
   void Reset();
}
```

A client that wishes to enumerate items over an enumerable asks for an enumerator from the enumerable using the `IEnumerator IEnumerable.GetEnumerator()` method. Then, the client uses methods `IEnumerator.MoveNext()` and `IEnumerator.Reset()` on the enumerator to iterate over items of the collection. An enumerator is a stateful object that keeps track of an index on the collection. A client calls the get accessor of the `IEnumerator.Current` property to access the item referenced currently by the index of the iterator. If we try to represent enumerables, enumerators and their clients in an UML diagram it would look like this:

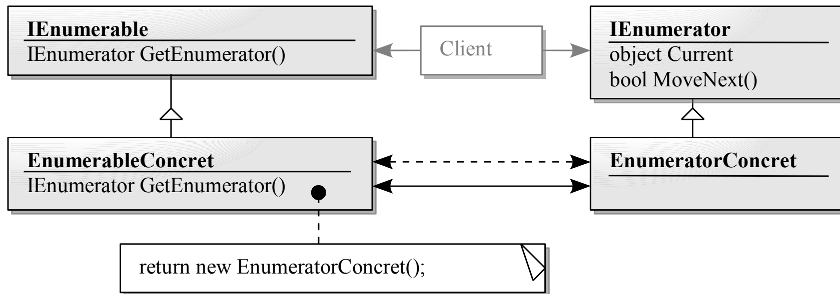


*Figure 14-5: UML diagram of the iterator design pattern*

We have just exactly described the *iterator design pattern* described in Gof as presented at page 854.

## An example

Here is an example of implementing iterators in C#1. The `Persons` class is the enumerable class. The `PersonsEnumerator` class is the enumerator class. Note that the `Persons` class could have played both roles of enumerable and enumerator if it would have implemented both interfaces. However, we'll see that it is always preferable that each role be implemented in a dedicated class (after all, we follow a central object oriented programming tenet: one responsibility/one class).

*Example 14-34*

```
public class Persons : System.Collections.IEnumerable {
   private class PersonsEnumerator : System.Collections.IEnumerator {
      private int index = -1;
      private Persons P;
      public PersonsEnumerator( Persons p ){ this.P = p; }
      public bool MoveNext() {
         index++;
         return index < P.m_Names.Length;
      }
      public void Reset() { index = -1; }
      public object Current { get { return P.m_Names[index]; } }
   }
   // The method GetEnumerator() of IEnumerable.
   public System.Collections.IEnumerator GetEnumerator() {
      return new PersonsEnumerator( this );
   }
   string[] m_Names;
   // The constructor to initialize the array.
   public Persons( params string[] Names ) {
      m_Names = new string[ Names.Length ];
      // Copy the array.
      Names.CopyTo( m_Names, 0 );
   }
```

*Example 14-34*

```
   // An indexer that returns the name from the index.
   private string this[ int index ] {
      get { return m_Names[index]; }
      set { m_Names[index] = value; }
   }
}
class Program {
   static void Main() {
      Persons arrPersons = new Persons(
                           "Michel", "Christine", "Mathieu", "Julien");
      foreach (string s in arrPersons)
         System.Console.WriteLine(s);
   }
}
```

It's a fairly long program to code such a simple functionality. Most of lines implement the indexer behavior of the enumerator. Hopefully, we'll see that C#2 dramatically reduces the amount of code to express the same need. This program outputs:

```
Michel
Christine
Mathieu
Julien
```

Note that the C# compiler has generated something like the following to interpret the `foreach` keyword:

*Example 14-35*

```
...
class Program {
   static void Main() {
      Persons arrPersons = new Persons(
                           "Michel", "Christine", "Mathieu", "Julien");
      System.Collections.IEnumerator e = arrPersons.GetEnumerator();
      while ( e.MoveNext() )
         System.Console.WriteLine( (string) e.Current );
   }
}
```

## *Several enumerators for a single enumerable*

An enumerable class is not required to implement the `IEnumerable` interface. This responsibility can be delegated to a third class `PersonsEnumerable`, for example:

*Example 14-36*

```
using System.Collections;
public class Persons // Doesn't implement IEnumerable!
   private class PersonsEnumerator : IEnumerator {
      ...
   }
   private class PersonsEnumerable : IEnumerable {
      private Persons m_Persons;
      internal PersonsEnumerable( Persons persons ) {
         m_Persons = persons;
      }
```

*Example 14-36*

```
      IEnumerator IEnumerable.GetEnumerator(){
         return new PersonsEnumerator( m_Persons );
      }
   }
   public IEnumerable InOrder{ get {return new PersonsEnumerable(this);} }
   ...
}
class Program {
   static void Main() {
      Persons arrPersons = new Persons(
                           "Michel", "Christine", "Mathieu", "Julien");
      foreach ( string s in arrPersons.InOrder )
         System.Console.WriteLine(s);
   }
}
```

This way, it is easy to implement several enumerator classes that target the same enumerable class. It can be quite useful to have an enumerator `Reverse` that yields elements in the reverse order, a `Shuffle` enumerator which traverses the list in a random order or an enumerator `EvenPosOnly` that returns only items at even positions.

### *Drawbacks of C#1 iterators*

Clearly, the C#1 syntax for iterators is too heavy compared to the functionality provided. For this reason, most developers don't use it. Moreover, with this syntax, trying to enumerate over items of a barely exotic collection (such as a binary tree) quickly becomes a nightmare.

## *C#2 iterators*

### *The keyword yield return*

C#2 has been enhanced with the `yield return` keyword in order to implement the iterator pattern seamlessly. Concretely, it relieves developers from the burden of implementing enumerator and enumerable classes. Here is our previous example rewritten:

*Example 14-37*

```
public class Persons : System.Collections.IEnumerable{
   string[] m_Names;
   public Persons( params string[] names ){
      m_Names = new string[names.Length];
      names.CopyTo(m_Names, 0);
   }
   // The GetEnumerator() method of IEnumerable.
   public System.Collections.IEnumerator GetEnumerator(){
      foreach (string s in m_Names)
         yield return s;
   }
}
class Program {
   static void Main() {
      Persons arrPersons = new Persons(
                      "Michel", "Christine", "Mathieu", "Julien");
      foreach (string s in arrPersons)
         System.Console.WriteLine(s);
   }
}
```

The behavior of the `yield return` keyword might stump you. It looks like the `yield return` keyword is returning a string but the type of the return value of the method `GetEnumerator()` is `IEnumerator`. Moreover, which class implements the `IEnumerator` returned since we don't explicitly supply such an implementation? We'll thoroughly shed light on these mysteries but it's still time to dig in C#2 iterators basics. Notice that a method can call the `yield return` keyword several times. For instance:

*Example 14-38*

```
public class Persons : System.Collections.IEnumerable {
   public System.Collections.IEnumerator GetEnumerator() {
      yield return "Michel";
      yield return "Christine";
      yield return "Mathieu";
      yield return "Julien";
   }
}
class Program {
   static void Main() {
      Persons arrPersons = new Persons();
      foreach (string s in arrPersons)
         System.Console.WriteLine(s);
   }
}
```

It seems that each time the thread calls `GetEnumerator()`, it branches itself just after the previous call to `yield return`. We'll soon verify that this hunch is the right one.

## Iterators and generics

In a language that supports generics, it would be a drag to still use the object's `IEnumerator.Current` property. Therefore, interfaces `IEnumerable` and `IEnumerator` are both provided in a generic form in the .NET 2 framework:

```
public interface System.Collections.Generic.IEnumerable<T> :
                                           System.Collections.IEnumerable {
   System.Collections.Generic.IEnumerator<T> GetEnumerator();
}
public interface System.Collections.Generic.IEnumerator<T> :
                       System.Collections.IEnumerator, System.IDisposable {
   T Current { get; }
}
```

Notice that the interface `IEnumerator<T>` implements the `IDisposable` interface. Notice also that the method `IEnumerator.Reset()` has been removed. Thanks to generics, we can now tell the compiler that our enumerable is a collection of strings instead of a collection of object:

*Example 14-39*

```
using System.Collections.Generic;
using System.Collections;
public class Persons : IEnumerable<string> {
   string[] m_Names;
   public Persons(params string[] names) {
      m_Names = new string[names.Length];
      names.CopyTo(m_Names, 0);
   }
   IEnumerator<string> IEnumerable<string>.GetEnumerator() {
      return PRIVGetEnumerator();
```

```
    }
    IEnumerator IEnumerable.GetEnumerator() {
       return PRIVGetEnumerator();
    }
    private IEnumerator<string> PRIVGetEnumerator() {
       foreach (string s in m_Names)
          yield return s;
    }
}
```

It is a shame to have to implement two versions of the `GetEnumeror()` method. This is a direct consequence of the fact the `IEnumerable<T>` implements `IEnumerable`. *Microsoft* engineers have made this choice to ensure that all enumerable be usable in a non generic form. In fact, several APIs only take an `IEnumerable` parameter while several APIs return an `IEnumerable<T>`. Without this trick, there would not be any implicit conversions between `IEnumerable<T>` and `IEnumerable` and developers would often have to explicitly specify such a conversion. This choice was made to the advantage to those who use enumerable classes rather than for those who develop them. As a developer, you will most often find your self in the first situation rather than the second.

## *Several enumerators for a single enumerable*

C#2 iterators are also a good mean to implement concisely several iterators on a single enumerable. For instance:

*Example 14-40*

```
public class Persons{
    string[] m_Names;
    public Persons( params string[] names ){
       m_Names = new string[names.Length];
       names.CopyTo(m_Names, 0);
    }
    public System.Collections.Generic.IEnumerable<string> Reverse {
       get {
          for (int i = m_Names.Length - 1; i >= 0; i--)
             yield return m_Names[i];
       }
    }
    public System.Collections.Generic.IEnumerable<string> PosEven {
       get {
          for (int i = 0; i < m_Names.Length ; i++,i++)
             yield return m_Names[i];
       }
    }
    public System.Collections.Generic.IEnumerable<string> Concat {
       get {
          foreach (string s in Reverse)
             yield return s;
          foreach (string s in PosEven)
             yield return s;
       }
    }
}
class Program {
    static void Main() {
       Persons arrPersons = new Persons(
```

*Example 14-40*

```
                     "Michel", "Christine", "Mathieu", "Julien");
      System.Console.WriteLine( "-->Iterator Reverse" );
      foreach ( string s in arrPersons.Reverse )
         System.Console.WriteLine( s );
      System.Console.WriteLine( "-->Iterator PosEven" );
      foreach ( string s in arrPersons.PosEven )
         System.Console.WriteLine( s );
      System.Console.WriteLine( "-->Iterator Concat" );
      foreach ( string s in arrPersons.Concat )
         System.Console.WriteLine( s );
   }
}
```

This program outputs:

```
-->Iterator Reverse
Julien
Mathieu
Christine
Michel
-->Iterator PosEven
Michel
Mathieu
-->Iterator Concat
Julien
Mathieu
Christine
Michel
Michel
Mathieu
```

## *The yield break keyword*

You might wish to enumerate a subset of an enumerable. In this case, the `yield break` keyword is the right way to tell the client that it should stop looping.

*Example 14-41*

```
...
   public IEnumerator<string> GetEnumerator() {
      for ( int i = 0; i < 2;i++ )
         yield return m_Names[i];
      yield break;
      // Warning : Unreachable code detected.
      System.Console.WriteLine("hello");
   }
...
```

This program outputs:

```
Michel
Christine
```

As a consequence, the code written after a `yield break` instruction is not reachable. The C# compiler emits a warning when it encounters such unreachable code.

## *Syntactic constraints on yield return and yield break keywords*

The `yield break` and `yield return` keywords can be used only inside the body of a method, the body of a property accessor or the body of an operator.

Whatever the kind of method that use `yield break` or `yield return` keywords, it must return one the following interfaces `System.Collections.Generic.IEnumerable<T>`, `System.Collections.IEnumerable`, `System.Collections.Generic.IEnumerator<T>` or `System.Collections.IEnumerator`.

`yield break` and `yield return` keywords can't be used in the body of an anonymous method.

`yield break` and `yield return` keywords can't be used in a `finally` block.

`yield break` and `yield return` keywords can't be used in a try block that has at least one `catch` block.

`yield break` and `yield return` keywords can't be used in a method that has `ref` or `out` arguments. More generally, a method that contains at least one of these keywords should not return any other information than the yielded item.

## *A recursive iterator example*

The following example shows the power of C#2 iterators by enumerating the items of a non-flat collection, such as a binary tree:

*Example 14-42*

```
using System.Collections.Generic;
public class Node<T> {
   public Node( T item , Node<T> leftNode , Node<T> rightNode ) {
      m_Item = item;
      m_LeftNode = leftNode;
      m_RightNode = rightNode;
   }
   public Node<T> m_LeftNode;
   public Node<T> m_RightNode;
   public T m_Item;
}
public class BinaryTree<T> {
   Node<T> m_Root;
   public BinaryTree( Node<T> root ){
      m_Root = root;
   }
   public IEnumerable<T> InOrder {
      get{
         return PrivateScanInOrder( m_Root );
      }
   }
   private IEnumerable<T> PrivateScanInOrder( Node<T> root ) {
      if ( root.m_LeftNode != null ) {
         foreach ( T item in PrivateScanInOrder( root.m_LeftNode ) ) {
            yield return item;
         }
      }
      yield return root.m_Item;
      if ( root.m_RightNode != null ) {
         foreach ( T item in PrivateScanInOrder( root.m_RightNode ) ) {
```

*Example 14-42*

```
            yield return item;
         }
      }
   }
}

class Program {
   static void Main() {
      BinaryTree<string> binaryTree = new BinaryTree<string> (
         new Node<string>( “A”,
            new Node<string>( “B” , null , null ),
            new Node<string>( “C” ,
               new Node<string>( “D” , null , null ),
               new Node<string>( “E” , null , null ) ) ) );
      foreach ( string s in binaryTree.InOrder )
         System.Console.WriteLine( s );
   }
}
```

A representation of the binary tree built by the `Main()` method is:

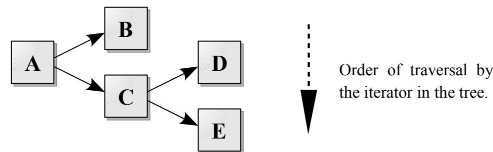


*Figure 14-6: Binary tree*

This program outputs:

```
B
A
D
C
E
```

# The C#2 compiler and iterators

If you are a curious developer, you might be delighted by the work done by the C#2 compiler to interpret `yield break` and `yield return` keywords. This is the subject of the current section. Keep in mind that iterators are syntactic sugar. Unlike generics, no IL instructions have been added to implement iterators.

## Enumerator classes are automatically built and used by the compiler

A method that uses at least one of the `yield break` or `yield return` keywords must return an enumerable or an enumerator (generic or not). In any cases, you can consider that an enumerator object is returned since the only purpose to implement an enumerable interface is to return an enumerator. If you’ve read the previous sections concerning anonymous methods, you might have guessed that for each method that contains at least one of the `yield break` or `yield return` keywords, the compiler builds a class that can be seen as a lexical environment.

This generated class implements the four interfaces System.Collections.Generic.IEnumerable<T>, System.Collections.IEnumerable, System.Collections.Generic.IEnumerator<T> and System.Collections.IEnumerator if the concerned method returns an enumerable object. The generated class implements only the two enumerator interfaces if the concerned method returns an enumerator object. Moreover, such a method is not directly compiled. The logic contained in its body is in the MoveNext() method of the generated class. Let's check all these facts on a small example:

*Example 14-43*

```
class Foo {
   public System.Collections.Generic.IEnumerable<string> AnIterator() {
      yield return “str1”;
      yield return “str2”;
      yield return “str3”;
   }
}
class Program {
   static void Main() {
      Foo collec = new Foo();
      foreach ( string s in collec.AnIterator() )
         System.Console.WriteLine( s );
   }
}
```

The following assembly is the compiled version of the previous program (the assembly is viewed with the tool *Reflector* introduced at page 19):

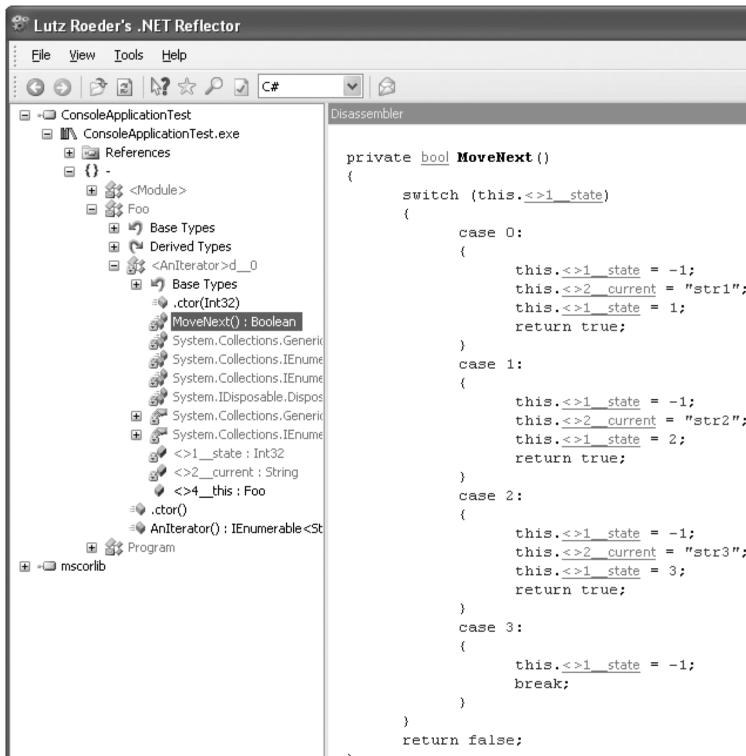


*Figure 14-7: C#2 compiler and iterators (1)*

To make things clear, here is the decompiled code of the Main() method. We can then check that an instance of the <AnIterator>d__0 class is created. In the previous section concerning anonymous methods, we said that such an instance is a closure. Thus in C#2, an iterator is a special kind of closure. Pay attention to the automatic and implicit use of the dispose pattern that surrounds the creation of the enumerator:

```
class Foo {
   public System.Collections.Generic.IEnumerable<string> AnIterator() {
      Foo.<AnIterator>d__0 d__1 = new Foo.<AnIterator>d__0(-2);
      d__1.<>4__this = this;
      return d__1;
   }
   ...
}
class Program {
   private static void Main() {
      Foo foo1 = new Foo();
      using (IEnumerator<string> enumerator1 =
         ( ( IEnumerator<string>)foo1.AnIterator().GetEnumerator() ) ) {
         while (enumerator1.MoveNext()) {
            string text1 = enumerator1.get_Current();
            Console.WriteLine(text1);
         }
      }
   }
}
```

Here is another example that deserves some decompilation:

*Example 14-44*

```
class Foo {
   public System.Collections.Generic.IEnumerable<int> AnIterator() {
      for ( int i = 0; i < 5; i++ ) {
         if( i == 3 ) yield break;
         yield return i;
      }
   }
}
class Program {
   static void Main() {
      Foo collec = new Foo();
      foreach ( int i in collec.AnIterator() )
         System.Console.WriteLine( i );
   }
}
```

The assembly exposed in Figure 14-8 is the compiled version of the previous program.

## Notes on generated classes

By looking at the bodies of the MoveNext() methods of the previous example, it seems that a state machine is built by the compiler when compiling a method that contains a yield break or yield return keyword. The state of the machine is held by the two instance fields <>1__state and <>2__current of the generated class. Remember that we observed that a thread executing an iterator is able to branch itself just after the previous call to yield return. This magic is possible thanks to the <>1__state field. A switch instruction on <>1__state is inserted at the beginning of the MoveNext() method while the value of <>1__state is properly set each time the running thread is about to leave the MoveNext() method. The field <>2__current holds the value computed by the previous call to MoveNext(). Thus the type of <>2__current is the type of enumerated elements (i.e. int32 in both previous examples).

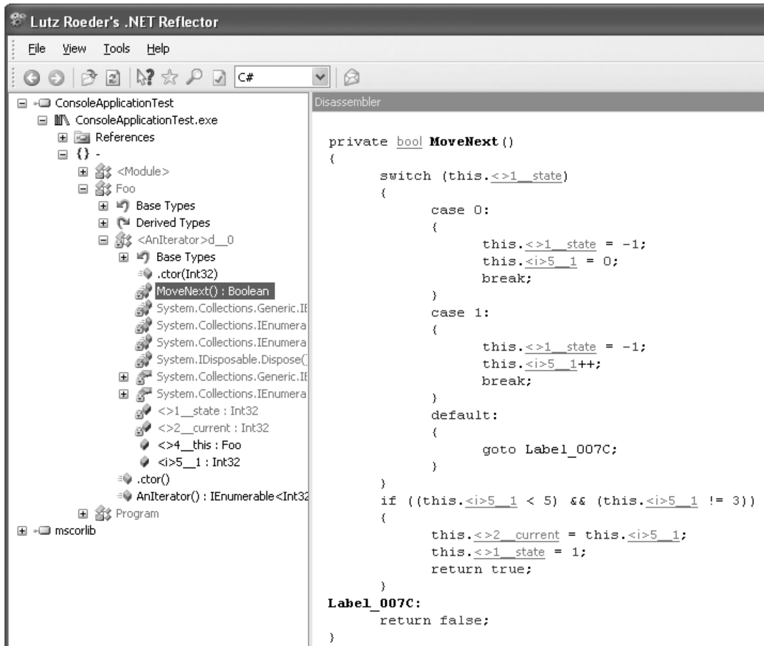


*Figure 14-8: C#2 compiler and iterators (2)*

## Notes on generated classes

By looking at the bodies of the `MoveNext()` methods of the previous example, it seems that a state machine is built by the compiler when compiling a method that contains a `yield break` or `yield return` keyword. The state of the machine is held by the two instance fields `<>1__state` and `<>2__current` of the generated class. Remember that we observed that a thread executing an iterator is able to branch itself just after the previous call to `yield return`. This magic is possible thanks to the `<>1__state` field. A switch instruction on `<>1__state` is inserted at the beginning of the `MoveNext()` method while the value of `<>1__state` is properly set each time the running thread is about to leave the `MoveNext()` method. The field `<>2__current` holds the value computed by the previous call to `MoveNext()`. Thus the type of `<>2__current` is the type of enumerated elements (i.e. int32 in both previous examples).

If the method that contains a `yield` instruction is an instance method, the generated class has an instance field called `<>4__this`. This field references the enumerable object.

Notice that if the method that contains a `yield` instruction has some local variables or some arguments, they are captured by the compiler. For instance, in the second example the local variable `i` is captured. As a consequence, the generated class has a field named `<i>5__1`.

# Advanced use of C#2 iterators

From the previous compilation analysis, we can infer two interesting differences between C#1 and C#2 iterators:

- By design, C#2 iterators support the *lazy evaluation pattern*. Concretely, the value of a yielded item is computed only when the client asks for it. When dealing with C#1 iterators, the values of all elements of an enumerable must have been computed before looping on it with a `foreach` instruction.

- C#2 iterators have no bounds on the number of items yielded. When dealing with C#1 iterators, the number of items had to be fixed before any looping with a `foreach` instruction.

We will take advantage of these properties in order to use iterators in other contexts that the iteration over the elements of an enumerable.

## *Definitions: coroutine and continuation*

A *coroutine* is a function that resumes its execution at the point where it stopped the last time it was called, as if nothing had happened between invocations. A *subroutine* is a function that starts its execution at the beginning of its body each time it is called. Clearly, all C# methods are subroutines except methods that contain a `yield` instruction that can be considered as coroutine.

Coroutines are a specialization of closures. Indeed, values of local variables and arguments must be captured by an object between invocations to make possible the magic of resuming at the point we quit last time. Notice also that the offset of the instruction in a coroutine body where the thread will resume its course, must also be captured between invocations. The closure that captures local variables values and the captured instruction offset is called a *continuation*. Thus, a continuation is an object created behind your back. Notice that a continuation is semantically equivalent to a thread's stack frame. This last remark will be soon relevant.

## *Harness the power of continuations and coroutines with iterators*

By coding two coroutines `A` and `B` where `A` calls `B` and `B` calls `A`, we can readily implement an infinite recursion that won't bloat the running thread's stack. This facility can be useful to develop a simplified chess game where the white treatment calls the black treatment and vice-versa. Suppose that some information must be stored by each treatment. In C#1, you would certainly design a class to store this information. In C#2 you can harness the power of continuations and coroutines to code this paradigm:

*Example 14-45*

```
using System;
using System.Collections;
public class Program {
   static IEnumerator White() {
      int whiteTreatmentInfo = 0;
      while ( true ) {
         Console.WriteLine( "white move, whiteTreatmentInfo=" +
                            whiteTreatmentInfo);
         whiteTreatmentInfo++;
         yield return black;
      }
   }
   static IEnumerator Black() {
      while ( true ) {
         Console.WriteLine( "black move" );
         yield return white;
      }
   }
   static IEnumerator black;
   static IEnumerator white;
   static void Main() {
      black = Black();
      white = White();
      IEnumerator enumerator = white; // Whites begin.
```

*Example 14-45*

```
        // We dispatche 5 times.
        for ( int i = 0; i < 5; i++ ) {
            enumerator.MoveNext();
            enumerator = (IEnumerator) enumerator.Current;
        }
    }
}
```

This program outputs:

```
white move, whiteTreatmentInfo=0
black move
white move, whiteTreatmentInfo=1
black move
white move, whiteTreatmentInfo=2
```

In this example, the role of the `black` and `white` static fields is essential. Each time the `White()` method is called a new continuation is created. Consequently the `White()` method must be called only one time. The field `white` is used to reference the unique continuation created by the method `White()`. The same remark also works with the field black and the method `Black()`.

In C#, a `goto` instruction and the label it references must lie in the same method body. This example shows that you can consider coroutines/continuations as a mean to implement '`goto`' between methods, without any risk to bloat the thread stack.

## *The pipeline pattern*

Iterators are perfect to implement the *pipeline pattern*, the one that you have used so many times in your console windows. For instance:

*Example 14-46*

```
using System.Collections.Generic;
class Program{
    static public IEnumerable<int> PipelineIntRange( int begin, int end ) {
        System.Diagnostics.Debug.Assert( begin < end );
        for( int i=begin; i<=end ; i++ )
            yield return i;
    }
    static public IEnumerable<int> PipelineMultiply( int factor ,
                                            IEnumerable<int> input ) {
        foreach ( int i in input )
            yield return i * factor;
    }
    static public IEnumerable<int> PipelineFilterModulo( int modulo ,
                                            IEnumerable<int> input ) {
        foreach ( int i in input )
            if( i%modulo == 0 )
                yield return i;
    }
    static public IEnumerable<int> PipelineJoin( IEnumerable<int> input1,
                                            IEnumerable<int> input2 ) {
        foreach ( int i in input1 )
            yield return i;
        foreach ( int i in input2 )
            yield return i;
```

*Example 14-46*

```
   }
   static void Main(){
      foreach (int i in PipelineJoin(
         PipelineIntRange(-4, -2), PipelineFilterModulo( 3,
                                       PipelineMultiply( 2,
                                          PipelineIntRange(1, 10) ) ) ) )
         System.Console.WriteLine(i);
   }
}
```

This program outputs:

```
-4
-3
-2
6
12
18
```

Values -4,-3 and -2 are readily understandable. Here is a schema that sheds light on values 6, 12 and 18:

```
PipelineIntRange(1,10) yields      1  2  3  4  5  6  7  8  9  10
PipelineMultiply(2) yields         2  4  6  8  10 12 14 16 18 20
PipelineFilterModulo(3) yields           6        12       18
```

If you modify `PipelineIntRange` like this…

*Example 14-47*

```
...
   static public IEnumerable<int> PipelineIntRange( int begin, int end ) {
      System.Diagnostics.Debug.Assert( begin < end );
      for ( int i = begin; i <= end; i++ ) {
         System.Console.WriteLine( "Yield:" + i );
         yield return i;
      }
   }
...
using System.Collections.Generic;
class Program{
   static public IEnumerable<int> PipelineIntRange( int begin, int end ) {
      System.Diagnostics.Debug.Assert( begin < end );
      for (int i = begin; i <= end; i++) {
         System.Console.WriteLine( "Yield:" + i );
         yield return i;
      }
   }
   static public IEnumerable<int> PipelineMultiply( int factor ,
                                                   IEnumerable<int> input ) {
      foreach ( int i in input )
         yield return i * factor;
   }
   static public IEnumerable<int> PipelineFilterModulo( int modulo ,
                                                   IEnumerable<int> input  ){
      foreach ( int i in input )
         if( i%modulo == 0 )
            yield return i;
```

*Example 14-47*

```
    }
    static public IEnumerable<int> PipelineJoin( IEnumerable<int> input1,
                                                 IEnumerable<int> input2) {
       foreach ( int i in input1 )
          yield return i;
       foreach ( int i in input2 )
          yield return i;
    }
    static void Main(){
       foreach (int i in PipelineJoin(
          PipelineIntRange(-4, -2), PipelineFilterModulo( 3,
                                       PipelineMultiply( 2,
                                          PipelineIntRange(1, 10) ) ) ) )
          System.Console.WriteLine(i);
    }
 }
```

…you get the following output:

```
Yield:-4
-4
Yield:-3
-3
Yield:-2
-2
Yield:1
Yield:2
Yield:3
6
Yield:4
Yield:5
Yield:6
12
Yield:7
Yield:8
Yield:9
18
Yield:10
```

This experience exposes the fact that items yield by an iterator are never stored somehow. As soon as an integer is yielded by `PipelineIntegerRange`, it is consumed by chained pipelines.

## Continuation vs. Threading

The pipeline pattern can be seen as a way to implement the *producer/consumer* paradigm. We generally use this paradigm in a multithreaded environment. This is the second time that threads are quoted in the current section on iterators. Indeed, we saw that information bundled in a continuation is the same as those contained in a thread stack frame.

The point is that continuations can help to implement some concurrency pattern. In the following example, a producer thread is computing *Fibonacci* numbers while the main thread is a consumer that prints yielded numbers on the console:

*Example 14-48*

```
using System.Collections;
using System.Threading;
public class Program {
   static AutoResetEvent eventProducterDone = new AutoResetEvent( false );
   static AutoResetEvent eventConsumerDone  = new AutoResetEvent( false );
   static int currentFibo;
   static void Fibo() {
      int i1 = 1;
      int i2 = 1;
      currentFibo = 0;
      // The producer triggers the cascade.
      eventProducterDone.Set();
      while( true ) {
         // Wait that the consumer is done.
         eventConsumerDone.WaitOne();
         // Let's build a new Fibonacci number.
         currentFibo = i1 + i2;
         i1 = i2;
         i2 = currentFibo;
         // Let's signal that the new number is ready to be consummed.
         eventProducterDone.Set();
      }
   }
   static void Main() {
      Thread threadProducteur = new Thread(Fibo);
      threadProducteur.Start();
      for ( int i = 1; i < 10; i++ ) {
         // Wait that the producer is done.
         eventProducterDone.WaitOne();
         // Let's consumme.
         System.Console.WriteLine(currentFibo);
         // Let's signal that we have consummed.
         eventConsumerDone.Set();
      }
   }
}
```

Notice that the state of the producer thread (i.e. values of `i1` and `i2`) is permanently stored on its own stack. Here is the same problematic implemented with a single thread and an iterator. This time, values of `i1` and `i2` are permanently stored in the continuation that is created when calling the `Fibo()` method.

*Example 14-49*

```
using System.Collections.Generic;
public class Program {
   static IEnumerator<int> Fibo() {
      int i1 = 1;
      int i2 = 1;
      int currentFibo = 0;
      while ( true ) {
         currentFibo = i1 + i2;
         i1 = i2;
         i2 = currentFibo;
         // Let's signal that the new number is ready to be consummed.
         yield return currentFibo;
```

*Example 14-49*

```
        }
    }
    static void Main() {
        IEnumerator<int> e = Fibo();
        for ( int i = 1; i < 10; i++ ) {
            // Let's the producer do its job.
            e.MoveNext();
            // Let's consumme.
            System.Console.WriteLine( e.Current );
        }
    }
}
```

This time the `i1` and `i2` values are all stored in the enumerator created by a call to the `Fibo()` method.

## A limitation of C#2 iterators

Clearly, the limitation that we underline in the present section won't hamper you in most of your algorithms. It can be found while trying to do something like a 'recursive iterator'. The idea was to use an iterator that calls itself to implement the *sieve of Eratosthène*. This sieve is an algorithm that allows computing prime numbers. It is explained by the following schema:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 |
  3   5   7   9    11    13    15    17 | 3 is prime,
                                        | 4,6,8,10,12,14,16,18 and
                                        | 20 are multiples of 2.
      5   7        11    13          17 | 5 is prime, 9 and 21
                                        | are multiples of 3.
          7        11    13          17 | 7 is prime, the list doesn't
                                        | contain any multiple of 5.
                   11    13          17 | 11 is prime, the list doesn't
                                        | contain any multiple of 7.
                         13          17 | 13 is prime, the list doesn't
                                        | contain any multiple of 11.
                                     17 | 17 is prime, the list doesn't
                                        | contain any multiple of 13.
```

The first number of each step is a prime. We remove this first number and its multiples to reach the next step. Each step yields a prime number.

The idea is to use as many pipelines as prime numbers we want. The first element of the pipeline will yield integers between 2 and N. In order to asses the number of prime numbers between 1 and N, we use the following theorem that was conjectured by *Gauss* and that was proved by *de la Vallée Poussin*. This theorem says that if `P(N)` is the number of prime numbers smaller than `N` then we have the approximation:

$$P(n) \approx \frac{n}{\ln n}$$

Here is the program:

*Example 14-50*

```
using System;
using System.Collections.Generic;
class Program {
```

*Example 14-50*

```
   static public IEnumerable<int> PipelineIntRange( int begin, int end ) {
      System.Diagnostics.Debug.Assert( begin < end );
      for ( int i = begin; i <= end; i++ )
         yield return i;
   }
   static public IEnumerable<int> PipelinePrime( IEnumerable<int> input ) {
      using ( IEnumerator<int> e = input.GetEnumerator() ) {
         e.MoveNext();
         int prime = e.Current;
         // The first number of the list is a prime.
         Console.WriteLine( prime );
         if ( prime != 0 ) {
            while ( e.MoveNext() ) {
               // Remove all multiple of the found prime.
               if ( e.Current % prime != 0 )
                  yield return e.Current;
            }
         }
      }
   }
   const int N = 100;
   static void Main() {
      // Apply the approximation of Gauss/de la Vallée Poussin
      // to get the number of iterators.
      int N_PRIME = (int)Math.Floor( ((double)N)/Math.Log(N) );

      // Build a pipeline of N_PRIME PipelineIntegerRange chained with
      // a PipelineIntegerRange. Each call to PipelinePrime yield an
      // iterator object that we store.
      List<IEnumerable<int>> list = new List<IEnumerable<int>>();
      list.Add(PipelinePrime( PipelineIntRange( 2, N ) ) );
      for( int i=1 ; i<N_PRIME ; i++ )
         list.Add( PipelinePrime(list[i-1]) );

      // Cascade the computation among iterators by yielding every
      // numbers between 2 and N.
      foreach ( int i in list[N_PRIME-1] );
   }
}
```

The limitation of C#2 iterators that is pinpointed here is the impossibility for an iterator object to reference and call itself. We can't use the keyword `this` in the body of the method that contains a `yield` instruction. In the case of an instance method, the keyword `this` references the current object. In the case of a static method, the compiler emits an error. It's like if the developer is not supposed to know what happens behind the scene. We could use reflection but this solution wouldn't be satisfactory.