IL DASM Tutorial

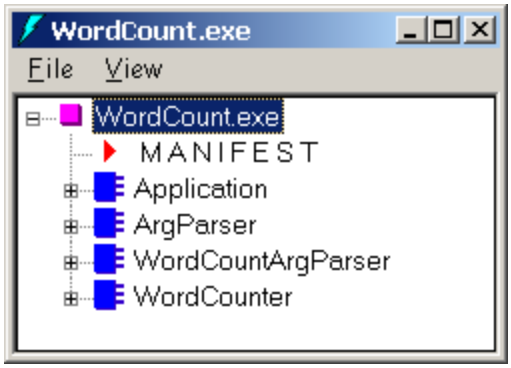
This paper offers an introduction to the ILDasm.exe tool that ships with the NGWS SDK. The ILDasm tool parses any NGWS EXE/DLL module and shows the information in a human-readable format. ILDasm – which stands for Intermediate Language (IL) dissasembler – shows more than just the IL code: it also displays namespaces and types, including their interfaces. You can use it to examine NGWS modules– such as MSCorLib.dll – as well as NGWS modules provided by others or ones you create yourself. Most NGWS developers will find ILDasm indispensable.

To work your way through this tutorial, we will be using the WordCount sample that ships with the SDK. We will use ILDasm to examine the WordCount.exe module.

To get started, build the WordCount sample and load it into ILDasm using the following command-line:

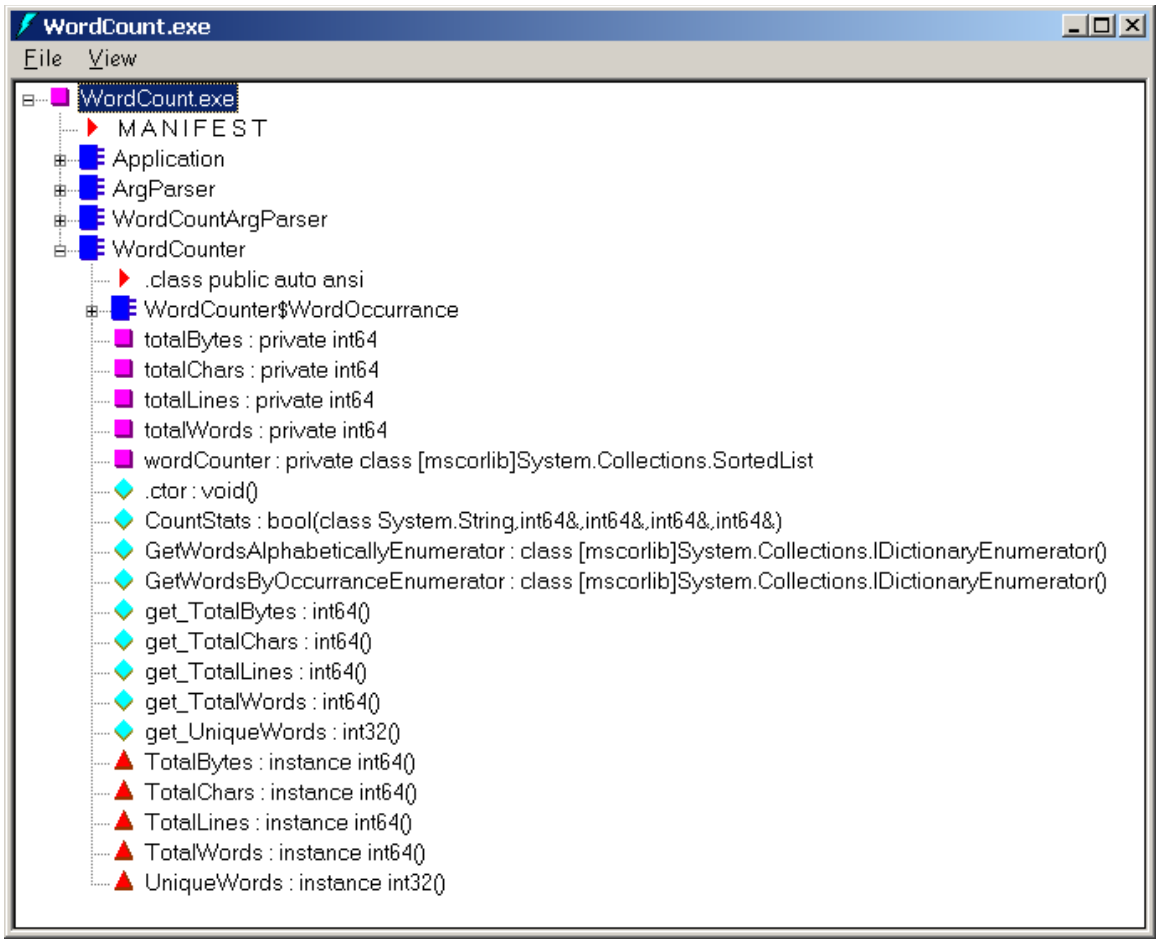
ILDasm WordCount.exe

This causes ILDasm's window to appear as shown below:



The tree in this window shows that manifest information contained inside WordCount.exe and the set of global class types: Application, ArgParser, WordCountArgParser, and WordCounter.

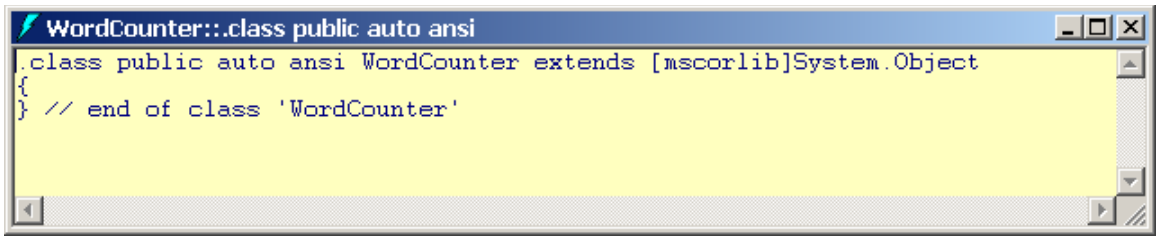
By double-clicking on any of the types in the tree, you can see more information about the type. In the figure below, I've expanded the WordCounter class type:



Here you can see all of WordCounter's members. The table below explains what each graphic symbol means:

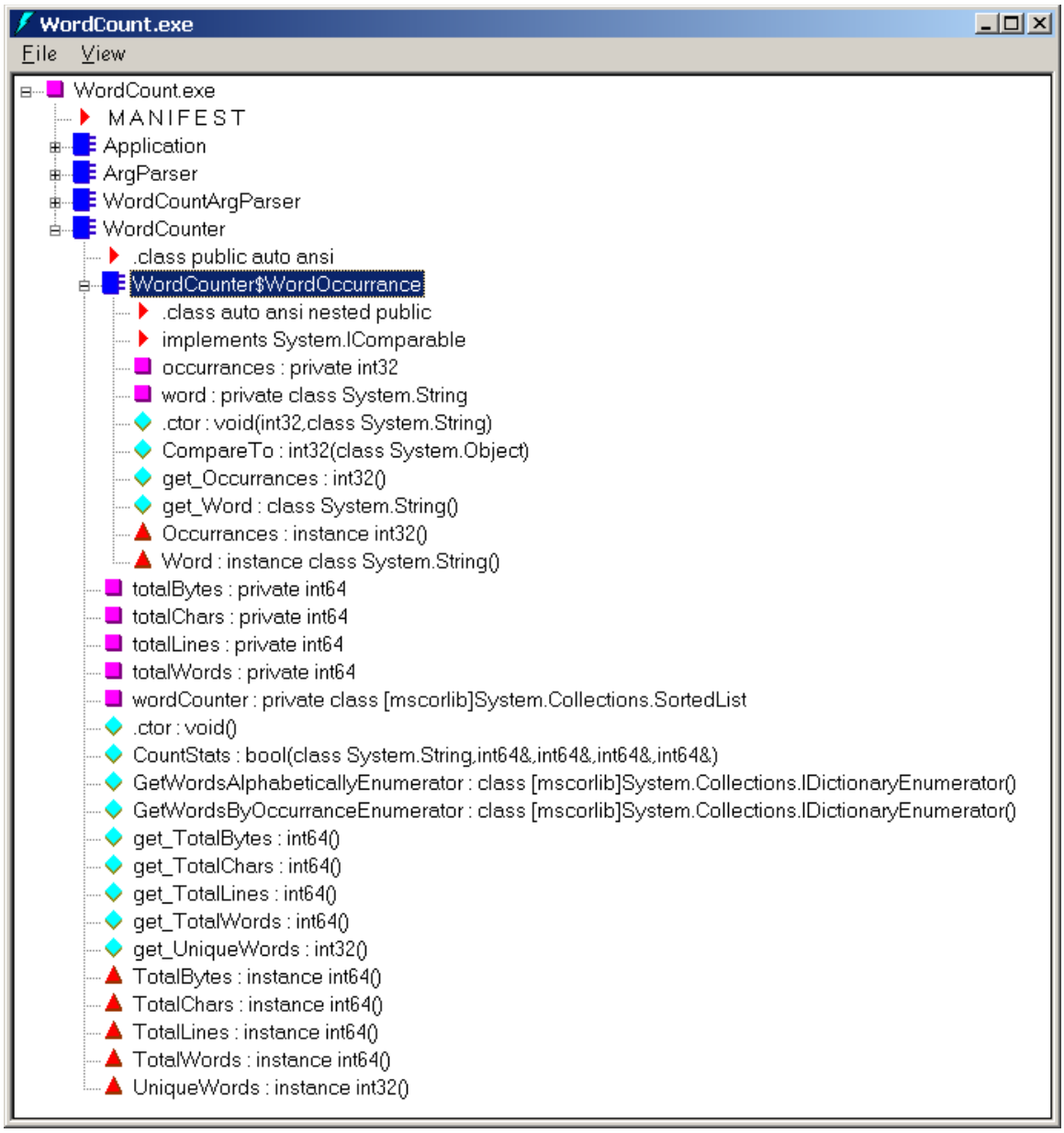
Symbol	Meaning
	More info
	Namespace
	Class
	Value type
	Interface
	Field
	Static field
	Method
	Static method
	Event
	Property

Double-clicking the ".class public auto ansi" entry shows the following information:



Here, you can easily see that the WordCounter type is derived from the System.Object type.

The WordCounter$WordOccurrance entry indicates a nested type. That is, the WordCounter type contains another type called WordOccurrence. You can expand this type to see its members as shown here:

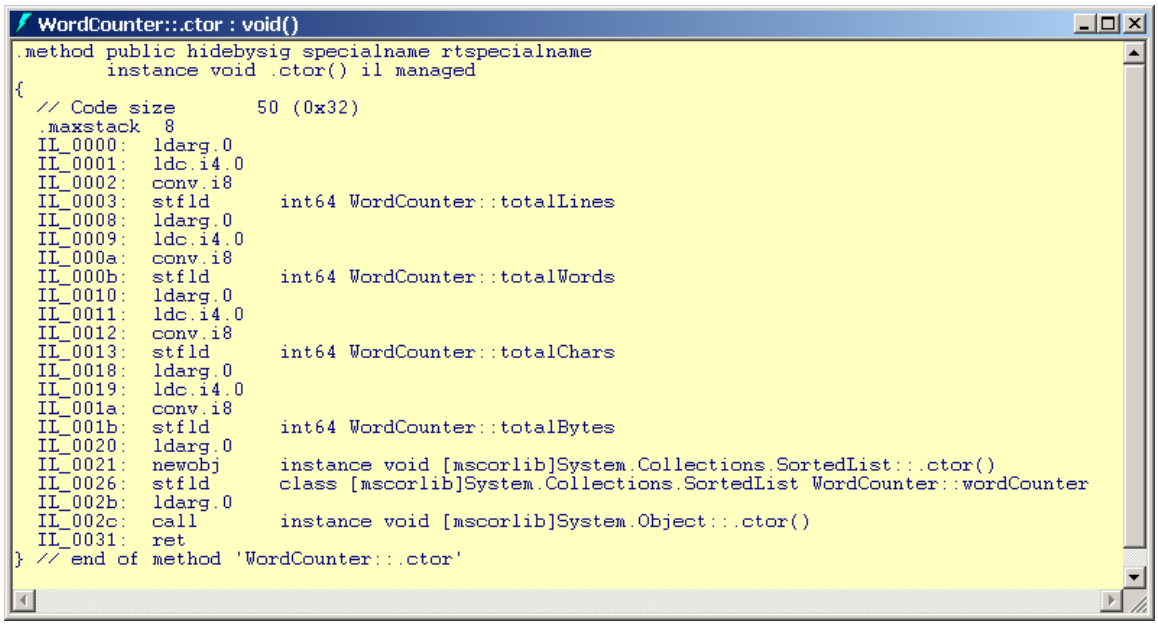


Looking at the tree, you can see that WordOccurrance implements the System.IComparable interface; specifically, the CompareTo method. But, for the rest of this conversation, let's ignore the WordOccurrance type and concentrate on the WordCounter type instead.

We see that the WordCounter type contains five private fields: totalBytes, totalChars, totalLines, totalWords, and wordCounter. The first four of these fields are instances of the int64 type while the wordCounter field is a reference to a System.Collections.SortedList type.

Following the fields, we see the methods. The first method, .ctor, is a constructor. This particular type has just one constructor but other types may have several constructors each with a different signature. WordCounter's constructor has a return type of void (as

do all constructors) and accepts no parameters. If you double-click on the constructor method, a new window appears showing the IL (intermediate language) contained within the method:



IL code is actually quite easy to read and understand. (For all the details, please see the IL Instruction Set Specification in the SDK documentation.) Towards the top, we see that the code for this constructor requires 50 bytes of IL. From this, we really have no idea how much native code will be emitted by the JIT compiler since this really depends on the host CPU and which compiler is being used to generate the code).

The IL virtual machine is stack based. So, in order to perform any operation, the operands are first pushed onto a virtual stack and then the operator executes. The operator grabs the operands off the stack, performs the desired operation and places the result back on the stack. At any one time, this method will have no more than 8 operands pushed onto the virtual stack. We can tell this by looking at the ".maxstack" attribute that appears just before the IL code.

Let's examine the first few IL instructions:

```
IL_0000:  ldarg.0       ; Load the object's 'this' pointer on the stack
IL_0001:  ldc.i4.0      ; Load the constant 4-byte value of 0 on the stack
IL_0002:  conv.i8       ; Convert the 4-byte 0 to an 8-byte 0
IL_0003:  stfld     int64 WordCounter::totalLines
```

The instruction at IL_0000 loads the first parameter passed to the method onto the virtual stack. Every instance method is always passed the address of the object's memory. This argument is called argument 0 and is never explicitly shown in the method's signature. So, even though the .ctor method looks like it receives 0 arguments, it actually receives 1 argument.

The instruction at IL_0000 loads this method's argument 0 (the pointer to this object) on to the virtual stack.

The instruction at IL_0001 loads a constant 4-byte value of 0 onto the virtual stack.

The instruction at IL_0002 takes the value on the top of the stack (the 4-byte 0) and converts it to an 8-byte 0 placing this 8-byte 0 on the stack.

At this point, the stack contains two operands: the 8-byte 0 and the pointer to this object. The instruction at IL_0003 uses both of these operands to store the value on the top of the stack (the 8-byte 0) into the totalLines field of the object identified on the stack.

The same IL instruction sequence is repeated for the totalChars, totalLines, and totalWords fields.

Initialization of the wordCounter field begins with instruction IL_0020:

```
IL_0020:  ldarg.0
IL_0021:  newobj     instance void [mscorlib]System.Collections.SortedList::.ctor()
IL_0026:  stfld      class [mscorlib]System.Collections.SortedList
WordCounter::wordCounter
```

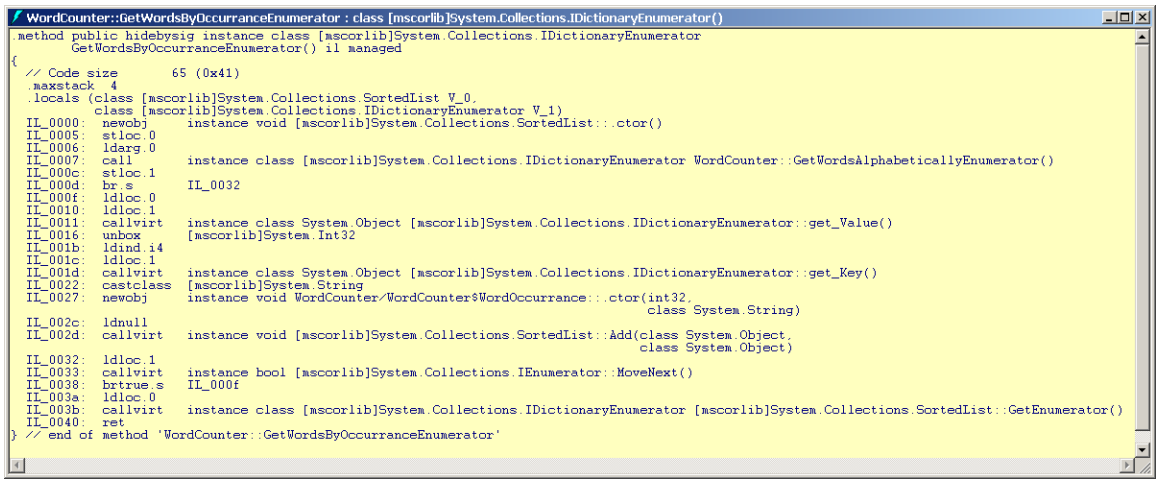
The instruction on IL_0020 pushes the 'this' pointer for our WordCounter on the virtual stack. This operand is not used by the newobj instruction but will be used by the stfld instruction at IL_0026.

The instruction at IL_0021 tells the runtime to create a new System.Collections.SortedList object and call its constructor passing no arguments. When newobj returns, the address of the SortedList object is on the stack. At this point, the stfld instruction at IL_0026 stores the pointer to the SortedList object in the WordCounter's wordCounter field.

After all of the WorkCounter's fields have been initialized, the instruction at IL_002b pushes the 'this' pointer on to the virtual stack and then calls the constructor in the base type (System.Object).

Of course, the last instruction at IL_0031 is the return instruction that causes WordCounter's constructor to return to the code that created it. Constructors have to return void so nothing is placed on the stack before the constructor returns.

Let's look at another example. Double-click on the GetWordsByOccurranceEnumerator method to see its IL.



We see that the code for this method is 65 bytes in size and that the method requires 4 slots on the virtual stack. In addition, this method has two local variables: one of them is of the System.Collection.SortedList type and the other is of the System.Collections.IDictionaryEnumerator type. Note that variable names mentioned in the source code are not emitted to the IL code. So, the variable names V_0 and V_1 are used instead.

When this method begins execution, the first thing it does is execute the newobj instruction which creates a new System.Collections.SortedList object and calls this object's default constructor. When newobj returns, the address of the created object is on the virtual stack. The stloc.0 instruction (IL_0005) stores this value in local variable 0, V_0 (which is of type System.Collections.SortedList).

At instructions IL_0006 and IL_0007, the WordCounter's this pointer (argument 0 passed to the method) is loaded on to the stack and then the GetWordsAlphabeticallyEnumerator method is called. When the call instruction returns, the address of the enumerator is on the stack. The stloc.1 instruction (IL_000c) saves this address in local variable 1, V_1 (which is of type System.Collections.IDictionaryEnumerator).

The br.s instruction at IL_000d causes an unconditional branch to the test condition IL of the while statement. This test condition IL begins at instruction IL_0032. At IL_0032, the address of V_1 (the IDictionaryEnumerator) is pushed on the stack and its MoveNext method is called. If MoveNext returns true, then an entry exists to be enumerated and the brtrue.s instruction jumpts to the instruction at IL_000f.

At instructions IL_000f and IL_0010 the addresses of the objects in V_0 and V_1 are pushed on the stack. Then, the IDictionaryEnumerator's get_Value property method is called to get the number of occurrences of the current entry. This number is a 32-bit value stored in a System.Int32 object. The code casts this Int32 object to an int value type. Casting a reference type to a value type requires the unbox instruction at IL_0016. When unbox returns, the address of the unboxed value is on the stack. The ldind.i4 instruction (IL_001b) loads the 4-byte value pointed at the address currently on the stack onto the stack. In other words, the unboxed 4-byte integer is placed on the stack.

At instruction IL_001c, the value of V_1 (the address of the IDictionaryEnumerator) is pushed on the stack and its get_Key property method is called. When get_Key returns, the address of the System.Object is on the stack. Our code knows that the dictionary contains strings and so the compiler casts this Object to a String using the castclass instruction at IL_0022.

The next few instructions (from IL_0027 to IL_002d inclusive) create a new WordOccurrance object and pass the address of it to the SortedLists's Add method.

At instruction IL_0032, the test condition of the while statement is evaluated again. If MoveNext returns true, the loop executes another iteration. However, if MoveNext returns false, then we fall through the loop and end up at instruction IL_003a. The instructions from IL_003a to IL_0040 call the SortLists's GetEnumerator method passing. The value returned is a System.Collections.IdictionaryEnumerator, which is left on the stack to become GetWordsByOccurranceEnumerator's return value.