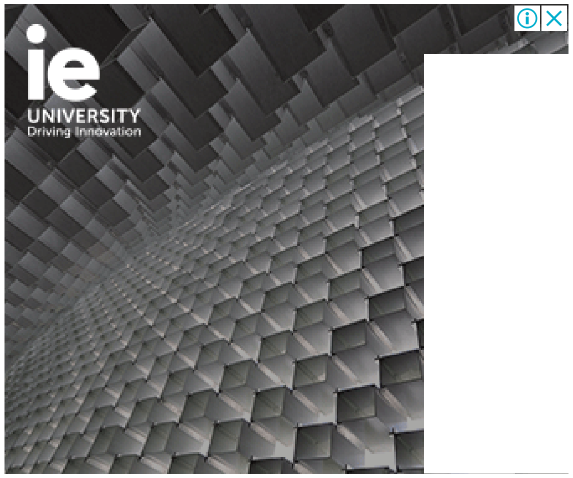
MSIL in Depth

MSIL in Depth



Here are some basic facts about MSIL programming. The content of an MSIL program is case sensitive. MSIL is also a freeform language. Statements can span multiple lines of code, in which lines can be broken at the white space. Statements are not terminated with a semicolon. Comments are the same as in the C# language. Double slashes (//) are used for single-line comments, and "/* *comment* */" is used for multiline comments. Code labels are colonterminated and reference the next instruction. Code labels must be unique within the scope in which it is defined.

In addition to the evaluation stack, the other important elements of a MSIL application are directives and the actual MSIL source code. Directives are dot-prefixed and are the declarations of the MSIL program. Source code is the executable content and control flow of the application.

Directives

There are several categories of directives. Assembly, class, and method directives are the most prominent. Assembly directives contain information that the compiler emits to the manifest, which is metadata pertaining to the overall assembly. Class directives define classes and the members of the class. This information is also emitted as standard metadata, which is data about types. Method directives define the particulars of a method, such as any local variables and the size of the evaluation stack.

Assembly Directives

Table 11-2 lists common assembly directives.

Table 11-2: Assembly Directives

Directive	Description

Directive	Description

.assembly	The *.assembly* directive defines the simple name of the assembly. The simple name does not include the extension. Assembly probing will uncover the correct extension. Adding the extension will cause normal probing to fail and manifest a binding exception when the assembly is referenced.
	
	This is the syntax of the *.assembly* directive:
	
	• .assembly *name {block}*
	
	The *assembly* block contains additional directives that further describe the assembly. These directives are optional. You need to provide only enough directives to uniquely identify the assembly. This is an *assembly* block with additional details:
	
	`.assembly Hello {`
	`.ver 1:0:0:0`
	`.locale "en.US"`
	`}`
	
	These are some of the directives available in the *assembly* block:
	
	• *.ver*—The four-part version number of the assembly
	• *.publickey*—The 8-byte public key token of the public/private key pair used to encrypt the hash of the assembly
	• *.locale*—The language and culture of the assembly
	• *.custom*—Custom attributes of the assembly

.assembly extern	The *.assembly extern* directive references an external assembly. The public types and methods of the referenced assembly are available to the current assembly. This is the syntax of the *.assembly extern* directive: • .assembly extern *name* as *aliasname* {*block*} The *as* clause is optional and for referencing assemblies that are similarly named but a different version, public key, or culture. Add the *.ver*, *.publickey*, *.locale*, and *.custom* directives to the *assembly extern* block to refine the identification of that assembly. Because of the importance of mscorlib.dll, the ILASM compiler automatically adds an external reference to that library. Therefore, *assembly extern mscorlib* is purely informative.
.file	The *.file* directive adds a file to the manifest of the assembly. This is useful for associating documents, such as a readme file, with an assembly. This is the syntax of the *.file* directive: • .file nometadata *file name* .hash = (*bytes*) *.entrypoint* The *file name* is the sole required element of the declaration. *Nometadata* is the primary option and stipulates that the file is unmanaged. `.file nometadata documentation.txt`

Directive	Description
.subsystem	The *.subsystem* directive indicates the subsystem used by the application, such as the graphical user interface (GUI) or console subsystem. This is distinct from the target type of the application, which is an executable, library, module, or so on. The ILASM compiler inserts this directive based on options specified when the application is compiled. You can also explicitly add this directive. This is the syntax of the *.subsystem* directive: • .subsystem *number* *Number* is a 32-bit integer in which: • 2 is a GUI application. • 3 is a console application.
.corflags	The *.corflags* directive sets the run-time flag in the CLI header. This defaults to 1, which stipulates an IL-only assembly. The corflags tool, introduced in .NET 2.0, allows the configuration of this flag. This is the syntax of the *.corflags* directive: • .corflags *flag* The flag is a 32-bit integer.
.stackreserve	The *.stackreserve* directive sets the stack size. The default size is 0x00100000. The following code calls *MethodA* iteratively. Without the *.stackreserve* directive, which defaults to 0x00100000, the *MethodA* method is called iteratively more than 110,000 times before exhausting the stack. Set the stack size to 0x0001000 using the *.stackreserve* directive. Now *MethodA* is called only about 21,000 times before quitting. Although the results may vary on your actual computer, the relative values are consistent.

Directive	Description
	```
	.assembly iterative {}
	.imagebase 0x00800000
	.stackreserve 0x00001000

	.namespace Donis.CSharpBook {
	    .class Starter {
	        .method static public void Main() il managed {
	            .entrypoint
	            ldc.i4.0
	            call void Donis.CSharpBook.Starter::
	                MethodA(int32)
	            ret
	        }

	        .method static public void MethodA(int32)
	                il managed {
	            ldarg.0
	            ldc.i4.1
	            add
	            dup
	            call void [mscorlib]
	                System.Console::WriteLine(int32)
	            call void Donis.CSharpBook.Starter::
	                MethodA(int32)
	            ret
	        }
	    }
	}
	```
.imagebase	The *.imagebase* directive sets the base address where the application is loaded. The default is 0x00400000. The load address of the application image and stack size is confirmable using the dumpbin tool. For example: • dumpbin /headers iterative.exe >iterative.txt

Class Directives

Table 11-3 describes the important class directives.

Table 11-3: Class Directives

Directive	Description

Directive	**Description**
.class header {*members*}	The *.class header* directive introduces a new reference type, value type, or interface into an assembly. The syntax of the *.class header* directive is as follows: *attributes classname* extends *basetype* implements *interfaces* There are a variety of attributes. This is a short list of the common attributes: • *abstract*—The type is abstract, and instances cannot be created. • *ansi* and *unicode*—Strings can be marshaled as ANSI or UNICODE. • *auto*—The memory layout of fields is controlled by the CLR. • *beforefieldinit*—Static methods are callable before the type is initialized. • *private* and *public*—Sets the visibility of the class outside of assembly. • *sealed*—The class cannot be inherited. • *serializable*—The contents of the class can be serialized. If the type inherits from another type, use the extends option. .NET supports only single class inheritance. The extends option is optional. If not present, the type inherits implicitly from *System.Object* or *System.ValueType*. The implements option lists the interfaces implemented by the type. The implements clause is optional and there are no default interfaces. The list of interfaces is comma-delimited. In the *members* block, members are declared with the appropriate directive: *.method*, *.field*, *.property*, and so on.
.custom constructorsignature	The *.custom* directive adds a custom attribute to the type.
.method	The *.method* directive defines a method. C# does not support global methods. Therefore, the *.method* directive is always included within a type. This is the syntax of the *.method* directive: • .method *attributes callingconv return methodname arguments implattributes { methodbody }* The *method* attributes are varied, including the accessibility attributes: *public*, *private*, *family*, and others. The default is *private*. Static methods have the *static* attribute, whereas instance methods possess the *instance* attribute. The default is an *instance* method. Here are additional attributes: • *final*—The method cannot be overridden. • *virtual*—The method is virtual.

Directive	Description

	• *hidebysig*—Hides the base class interface of this method. This flag is used only by the source language compiler.
	• *newslot*—Creates a new entry in the vtable for this method, which prevents overriding a method of the same name in a base class. For example, this option is used with the *add_Event* and *remove_Event* methods of an event.
	• *abstract*—The method has no implementation and is assumed to be implemented in a descendant.
	• *specialname*—The method is special, such as *get_Property* and *set_Property* methods. These methods are treated in a special way by tools.
	• *rtspecialname*—The method has a special name, such as a constructor. These methods are treated in a special way by the CLR.
	The calling convention pertains mostly to native code, in which a variety of calling conventions are supported: *fastcall*, *cdecl*, and others.
	The implementation attributes include the following:
	• *cil*—The method contains MSIL code.
	• *native*—The method contains platform-specific code.
	• *runtime*—The implementation of the method is provided by the CLR. When defining delegates, the delegate class and methods are generated by the run time.
	• *managed*—The implementation is managed.
	Here is the declaration of a C# method:
	`virtual public int MethodA(int param1, int param2)`
	This is the MSIL code for that same method:
	`.method public hidebysig newslot virtual`
	`instance int32 MethodA(int32 param1,`
	`int32 param2) cil managed`

Directive	Description

.field	The *.field* directive defines a new field, which contains state for a class or instance.
	
	The syntax of the *.field* directive is as follows:
	
	• .field *attributes type fieldname fieldinit* at *datalabel*
	
	The accessibility attributes are the same as described with methods. Fields can be assigned the *static* attribute but not the *instance* attribute. The default is an *instance* field.
	
	This is a list of other common field attributes:
	
	• *initonly*—Defines a *readonly* field.
	• *specialname*—The field is special.
	• *rtspecialname*—The field has a special name.
	
	The *fieldinit* and *datalabel* options are optional.
	
	This is a field defined in a C# class:
	
	`private readonly int fielda=10;`
	
	This is the same field translated to MSIL code. The compiler adds a no-argument constructor, where *fielda* is initialized to 10.
	
	`.field private initonly int32 fielda`

Directive	**Description**
.property	The *.property* directive introduces a property member to a class. It also declares the *get* and *set* methods associated with the property. This is the syntax of the *.property* directive: • .property *attributes return propertyname parameters default* {*propertyblock*} The attributes of a property can be *specialname* or *rtspecialname*. The *return* is the return type of the property. The composition of *propertyname* and *parameters* is the signature of the property. The *default* option sets the default value of the property. Within the *property* block, the *.get* directive declares the signature of the *get* method, whereas the *.set* directive declares the *set* method. The *.propertybody* includes only the method declarations. The *get* and *set* methods are actually implemented at the class level, not within the property. This is a property defined and implemented in a C# application: `public int propa { get { return 0; } }` This is the same property in MSIL code: `.property instance int32 propa() { .get instance int32 Donis.CSharpBook.Starter::get_propa() }`

C# code from the cell above:

```
public int propa {
 get {
 return 0;
 }
}
```

MSIL code from the cell above:

```
.property instance int32 propa()
{
 .get instance int32
 Donis.CSharpBook.Starter::get_propa()
}
```

Directive	**Description**

.event	The *.event* directive defines a new event in a class.
	
	This is the syntax of the *.event* directive:
	
	• .event *classref eventname* { *eventbody* }
	
	Classref is the underlying type of the event, such as *EventHandler*.
	
	The *.eventbody* directive encapsulates the *.addon* and *.removeon* directives. The *.addon* directive declares the method used to add subscribers. The *.removeon* directive declares the method for removing subscribers. The *add* and *remove* methods are implemented in the class and not the event.
	
	This is the C# code that declares an event:
	
	`public event EventHandler EventA;`
	
	Here is the MSIL code for that same event:
	
	`.event [mscorlib]System.EventHandler EventA`
	`{`
	` .addon instance void`
	` Donis.CSharpBook.Starter::add_EventA(`
	` class [mscorlib]System.EventHandler)`
	` .removeon instance void`
	` Donis.CSharpBook.Starter::remove_EventA(`
	` class [mscorlib]System.EventHandler)`
	`}`

Method Directives

The *.method* directive adds a method to a class. MSIL allows for global methods. Global methods break the rules of encapsulation and other tenets of OOP. For this reason, C# does not support global methods. MSIL generated from the C# compiler (csc) uses the *.method* directive solely to define member methods. The *method* block contains further directives and the implementation code (MSIL).

Table 11-4 lists the directives that are frequently included in the *method* block.

Table 11-4: Directives Included in the Method Block

Directive	**Description**

Directive	Description
.locals	The *.locals* directive declares local variables that are accessible using a symbolic name or index. Local variables form a zero-based array. This is the syntax of the *.locals* directive: • .locals^1 (*[index]local1, [index] local2, [index] localn*) • .locals^2 init (*[index]local1, [index] local2, [index] localn*) The *.locals*^{1} directive defines one or more local variables. Explicit indexes can be set for each local variable. By default, the local variables are indexed sequentially starting at zero. The *.locals*^{2} directive adds the *init* keyword, which requests that local variables be initialized to a zero-based value before the method executes. The *init* keyword is required to pass code verification. Therefore, the C# compiler only emits the *.locals*^{2} directive. Local variables do not have to be declared at the beginning of a method, and they can appear more than once in a method—each time declaring different local variables.
.maxstack	The *.maxstack* directive sets the number of slots available on the evaluation stack. Without this directive, the default is eight slots, which is the number of items that can be placed on the evaluation stack simultaneously. This is the syntax of the *.maxstack* directive: • .maxstack *slots*
.entrypoint	The *.entrypoint* directive designates a method as the entry point method of the application. This directive can appear anywhere in the method, but best practice places the *.entrypoint* directive at the start of the method. In C#, the entry point method is *Main*. In MSIL, any *static* method can be accorded this status.

The following program defines *MSILFunc* as the entry point method. The *.entrypoint* directive is found at the end of this method. The *.locals* directive defines two locals and assigns explicit indexes. Essentially, the normal indexes are reversed. The instruction *stloc.0* will update the second local variable. *MSILFunc* refers to the local variables both as symbolic names and indexes. The *MSILFunc* method returns *void*. In MSIL code, the *ret* instruction is required even when a function returns nothing. In C#, the return is optional for methods returning *void*. The method displays the values of 10 and then 5.

```
.assembly extern mscorlib {}
.assembly application {}

.namespace Donis.CSharpBook {

    .class Starter {

        .method static public void MSILFunc() il managed {
           .locals init ([1] int32 locala, [0] int32 localb)
           ldc.i4.5
           stloc.0
           ldc.i4 10
           stloc.1
           ldloc locala
           call void [mscorlib] System.Console::WriteLine(int32)
           ldloc localb
           call void [mscorlib] System.Console::WriteLine(int32)
           .entrypoint
           ret
          }
    }
}
```

MSIL Instructions

MSIL includes a full complement of instructions, many of which were demonstrated in previous examples. Each instruction is also assigned an opcode, which is commonly 1 or 2 bytes. 2-byte opcodes are always padded with a 0xFE byte in the high-order byte. Opcodes are often followed with operands. Opcodes, which provide an alternate means of defining MSIL instructions, are used primarily when emitting code dynamically at run time. The *ILGenerator .Emit* method records instructions using opcodes, which is in the *System.Reflection.Emit* namespace.

The byte option of ILDASM adds opcodes to the disassembly. The following is a partial listing of the hello.exe disassembly, which includes just the *Main* method. As ascertained from the disassembly, the opcode for *ldstr* is 72, the opcode for *stloc* is 0A, and the opcode for *call* is 28.

```
.method public static void Main() cil managed
{
    .entrypoint
    .maxstack 2
    .locals init (string V_0)
    IL_0000: /* 72 | (70)000001 */ ldstr "Donis"
    IL_0005: /* 0A | */ stloc.0
    IL_0006: /* 72 | (70)00000D */ ldstr "Hello, {0}!"
    IL_000b: /* FE0C | 0000 */ ldloc V_0
    IL_000f: /* 28 | (0A)000001 */ call void
    [mscorlib]System.Console::WriteLine(string, object)
    IL_0014: /* 2A | */ ret
}
```

Short Form Some MSIL instructions have normal and short-form syntax. The short forms of the instruction have a .s suffix. The short form of the *ldloc* instruction is *ldloc.s*. The short form of the *br* instruction is *br.s*. Normal instructions have 4-byte operands, and short-form instructions are limited to 1-byte operands.

When used injudiciously, the short-form syntax can cause unexpected results:

```
.assembly extern mscorlib {}
.assembly application {}

.namespace Donis.CSharpBook {

 .class Starter {
 .method static public void Main() il managed {
 .entrypoint
 ldc.i4.s 50000
 call void [mscorlib] System.Console::WriteLine(int32)
 ret
 }
 }
}
```

In the preceding application, a constant of 50000 is placed on the evaluation stack. However, the *ldc* instruction is in the short form. It is difficult to fit 50000 into a single byte, so the constant overflows the byte. For this reason, the application incorrectly displays *80*.

The next section of the book reviews the categories of MSIL instructions, such as branch, arithmetic, call, and array groups of instructions. Because of the prevalence of the evaluation stack, load and store instructions are the most frequently used of all MSIL instructions. That is a good place to start.

Load and Store Methods Load and store instructions transfer data between the evaluation stack and memory. Load commands push memory, such as a local variable, to the evaluation stack. Store commands move data from the evaluation stack to memory. Information placed on the evaluation stack is then consumed by method parameters, arithmetic operations, and other MSIL instructions. Data not otherwise consumed should be removed from the evaluation stack before the current method returns. The pop instruction is the best command to remove extraneous data from the evaluation stack. Data needed for an instruction should be placed on the evaluation stack immediately prior to the execution of that instruction. If not, an *InvalidProgramException* is triggered. Method returns are also placed on the evaluation stack.

Table 11-5 lists the basic load instructions.

Table 11-5: Load Instructions

Instruction	Description

Instruction	Description
ldc	The *ldc* instruction posts a constant to the evaluation stack, which can be an integral or floating-point value. This is the syntax of the *ldc* instruction: • ldc[1].*type value* • ldc[2].i4.*number* • ldc[3].i4.s *number* The *ldc*[1] instruction loads a constant of the specified type onto the evaluation stack. The *ldc*[2] instruction is more efficient and transfers an integral value of -1 and between 0 and 8 to the evaluation stack. The special format for -1 is *ldc.i4.m1*.
ldloc	The *ldloc* instruction copies the value of a local variable to the evaluation stack. This is the syntax of the *ldloc* instruction: • ldloc[1] *index* • ldloc[2].s *index* • ldloc[3] *name* • ldloc[4].s *name* • ldloc[5].*n* The *ldloc*[1] and *ldloc*[2] instructions use an index to identify a local variable, which is then loaded on the evaluation stack. The *ldloc*[3] and *ldloc*[4] instructions identify the local variable with the symbolic name. The short form of *ldloc* efficiently loads local variables from 4 to 255. The *ldloc*[5] instruction is optimized to load local variables from 0 to 3.
ldarg	The *ldarg* instruction places a method argument on the evaluation stack. This is the syntax of the *ldarg* instruction, which is identical to the *ldloc* instruction: • ldarg[1] *index* • ldarg[2].s *index* • ldarg[3] *name* • ldarg[4].s *name* • ldarg[5].*n*
ldnull	The *ldnull* instruction places a null on the evaluation stack. This instruction has no operands.

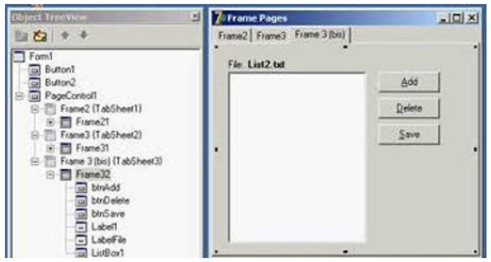
Table 11-6 lists the basic store instructions.

Table 11-6: Store Instructions

Instruction	**Description**
stloc	The *stloc* instruction transfers a value from the evaluation stack to a local variable. The value is then removed from the evaluation stack. This is the syntax of the *stloc* instruction, which is the same as the *ldloc* instruction: • stloc *index* • stloc.s *index* • stloc *name* • stloc.s *name* • stloc.*n*
starg	The *starg* instruction moves a value from the evaluation stack to a method argument. The value is then popped from the evaluation stack. This is the syntax of the *starg* instruction: • starg *num* • starg.s *num* The short form of the *starg* instruction is efficient for the first 256 arguments.

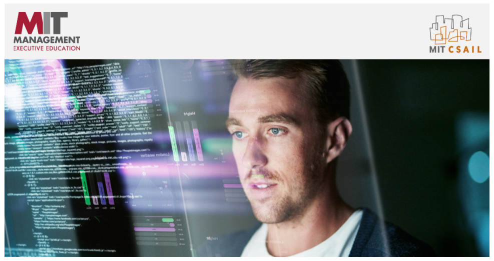
Игра длинною в жизнь 2019

Ad Xcraft 21+



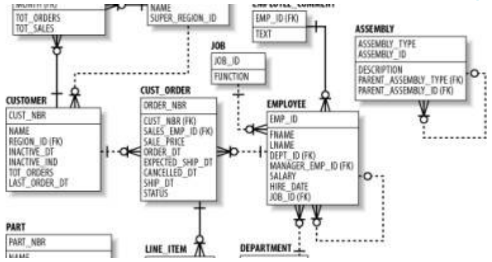
Understanding Frames

etutorials.org



Study machine learning

Ad MIT Sloan and MIT CSAIL



1.3 A Simple Database

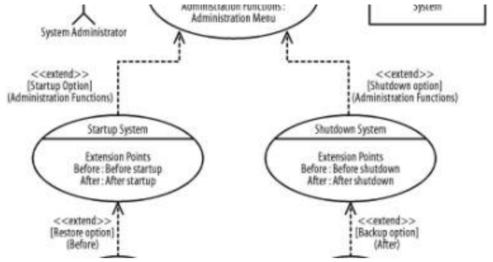
etutorials.org

Выбирай любимую расу

Ad Elvenar

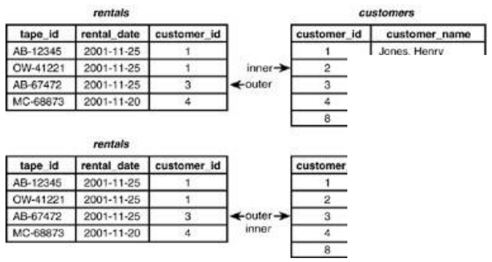
Web Database Applications

etutorials.org



4.4 Dependencies

etutorials.org



Understanding How PostgreSQL Executes

etutorials.org