Module 1

.NET runtime

.NET runtime - CIL, JIT, assembler

What is .NET...?

- *"open **specification** (technical standard) developed by Microsoft and standardized by ISO and Ecma that describes executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architecture"* - Wikipedia

What is .NET...?

- *"open **specification** (technical standard) developed by Microsoft and standardized by ISO and Ecma that describes executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architecture"* - Wikipedia

- This specification name is **Common Language Infrastructure**...

What is .NET...?

- *"open **specification** (technical standard) developed by Microsoft and standardized by ISO and Ecma that describes executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architecture"* - Wikipedia
- This specification name is **Common Language Infrastructure**...
- specified in *"ECMA-335. Common Intermediate Language (CIL)"*

What is .NET...?

- *"open **specification** (technical standard) developed by Microsoft and standardized by ISO and Ecma that describes executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architecture"* - Wikipedia
- This specification name is **Common Language Infrastructure**...
- specified in *"ECMA-335. Common Intermediate Language (CIL)"*
  do not confuse with *"ECMA-334. C# Language Specification."*

What is .NET...?

- *"open **specification** (technical standard) developed by Microsoft and standardized by ISO and Ecma that describes executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architecture"* - Wikipedia

- This specification name is **Common Language Infrastructure**...

- specified in *"ECMA-335. Common Intermediate Language (CIL)"*
  do not confuse with *"ECMA-334. C# Language Specification."*

- obviously, above that we have the whole *ecosystem*:
  - compilers - C# (Roslyn), F#, ...
  - libraries - Base Class Library, ...
  - frameworks - ASP.NET, Entity Framework, ...
  - tools - MSBuild, Visual Studio, NuGet, ...

What is .NET...?

Common Language Infrastructure:

What is .NET...?

Common Language Infrastructure:

- The Common Type System (**CTS**) - set of **data types**

What is .NET...?

Common Language Infrastructure:

- The Common Type System (**CTS**) - set of **data types**
- The Metadata - information about the **program structure/data**

What is .NET...?

Common Language Infrastructure:

- The Common Type System (**CTS**) - set of **data types**
- The Metadata - information about the **program structure/data**
- The Common Language Specification (**CLS**) - **set of rules** to interoperate with other *CLS-compliant* languages

What is .NET...?

Common Language Infrastructure:

- The Common Type System (**CTS**) - set of **data types**
- The Metadata - information about the **program structure/data**
- The Common Language Specification (**CLS**) - **set of rules** to interoperate with other *CLS-compliant* languages
- The Virtual Execution System (**VES**) - **loads and executes** CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime

What is .NET...?

Common Language Infrastructure:

- The Common Type System (**CTS**) - set of **data types**
- The Metadata - information about the **program structure/data**
- The Common Language Specification (**CLS**) - **set of rules** to interoperate with other *CLS-compliant* languages
- The Virtual Execution System (**VES**) - **loads and executes** CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime

And... CLI-compatible programs compiles to Common Intermediate Language (**CIL**) 🤯

What is .NET...?

Common Language Infrastructure:

- The Common Type System (**CTS**) - set of **data types**
- The Metadata - information about the **program structure/data**
- The Common Language Specification (**CLS**) - **set of rules** to interoperate with other *CLS-compliant* languages
- The Virtual Execution System (**VES**) - **loads and executes** CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime

And... CLI-compatible programs compiles to Common Intermediate Language (**CIL**) 🤯

(we will return to that)

Common Type System (CTS) vs Common Language Specification (CLS)...

- CLS is a subset of CTS!
- our code may by CLS-compliant or not... Eg. public field starting with underscore is not:

```
[assembly: CLSCompliant(true)]
namespace ConsoleApp
{
    0 references
    public class SomeType
    {
        public int _Field = 4;
    }
```

(field) int SomeType._Field

CS3008: Identifier '_Field' is not CLS-compliant

Show potential fixes (Ctrl+.)

- usually we don't care - 99% is written in C# anyway
- `uint` is also not-CLS compliant and so on, and so forth

What is .NET runtime aka Virtual Execution System (VES)?

- piece of code executing CIL code...
  - knowing about Common Type System and Metadata
  - we may ignore CLS :)

What is .NET runtime aka Virtual Execution System (VES)?

- piece of code executing CIL code…
  - knowing about Common Type System and Metadata
  - we may ignore CLS :)
- … from assembly/module
  - a physical container for code and metadata

Module, assembly, ...

ECMA-335, II.6:

- *"an **assembly** is a set of one or more files deployed as a unit, (...) contains **manifest** defining:*
    - *version, name, culture, ...*
    - *which other files, if any, belong to the assembly*

Module, assembly, ...

ECMA-335, II.6:

- *"an **assembly** is a set of one or more files deployed as a unit, (...) contains **manifest** defining:*
  - *version, name, culture, ...*
  - *which other files, if any, belong to the assembly*
- *a **module** is a single file containing executable content in the format specified here*"

Module, assembly, ...

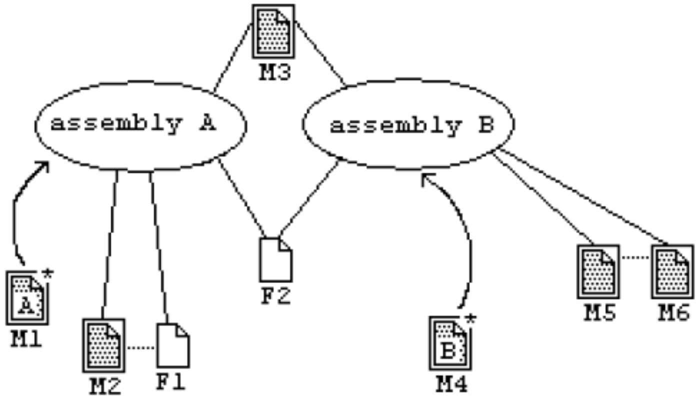
ECMA-335, II.6:

- *"an **assembly** is a set of one or more files deployed as a unit, (...) contains **manifest** defining:*
  - *version, name, culture, ...*
  - *which other files, if any, belong to the assembly*
- *a **module** is a single file containing executable content in the format specified here*"

II.6.2:

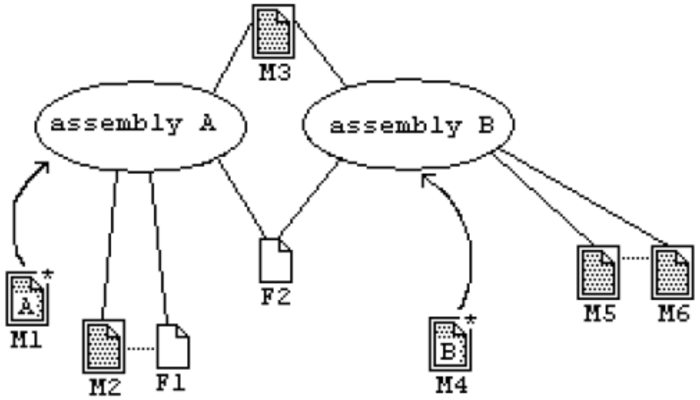
- *"assembly is specified as a module that contains a manifest in the metadata"*

Module, assembly, ...



(from ECMA-335)

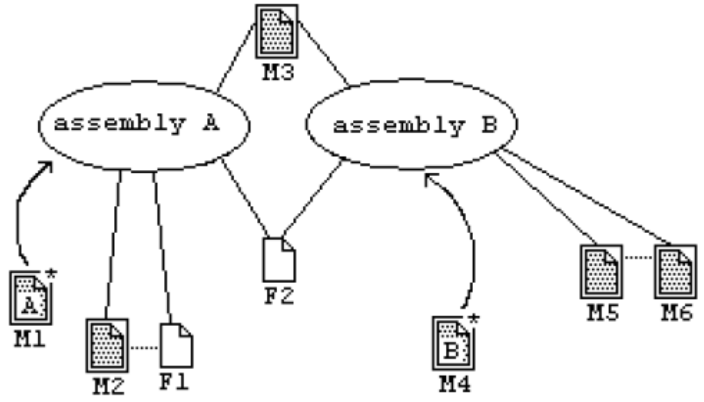
Module, assembly, ...



(from ECMA-335)

- **Fx** - resource files (like bitmaps)

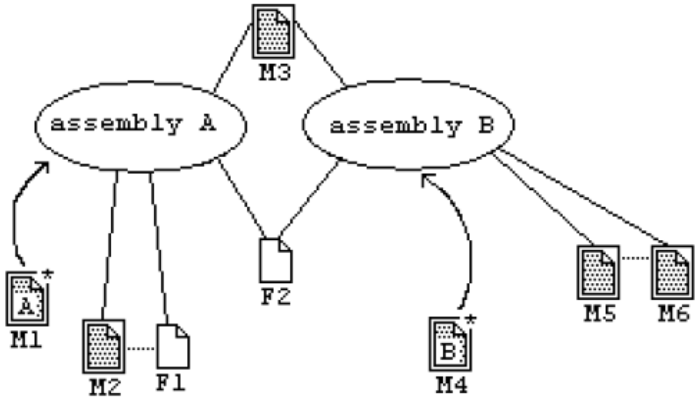
Module, assembly, ...



(from ECMA-335)

- `Fx` - resource files (like bitmaps)
- `Mx` - module files
  - `M1` and `M4` - contain manifest declaring assembly `A` and `B` (respectively)
  - the assembly declaration in `M1` and `M4` references other modules

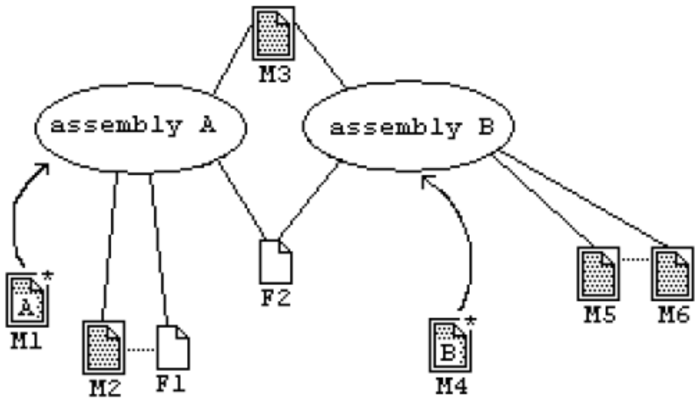
Module, assembly, ...



(from ECMA-335)

- `Fx` - resource files (like bitmaps)
- `Mx` - module files
  - `M1` and `M4` - contain manifest declaring assembly `A` and `B` (respectively)
  - the assembly declaration in `M1` and `M4` references other modules
- Usually, a module belongs only to one assembly, but it is possible to share it across assemblies.
  - When assembly `A` is loaded at runtime, an instance of `M3` will be loaded for it.
  - When assembly `B` is loaded into the same application domain, (...) `M3` will be shared for both assemblies.

Module, assembly, ...



(from ECMA-335)

- `Fx` - resource files (like bitmaps)
- `Mx` - module files
    - `M1` and `M4` - contain manifest declaring assembly `A` and `B` (respectively)
    - the assembly declaration in `M1` and `M4` references other modules
- Usually, a module belongs only to one assembly, but it is possible to share it across assemblies.
    - When assembly `A` is loaded at runtime, an instance of `M3` will be loaded for it.
    - When assembly `B` is loaded into the same application domain, (...) `M3` will be shared for both assemblies.
- *"The Visual Studio IDE for C# and Visual Basic can only be used to create single-file assemblies."*
    - so, just a regular one-to-one correspondence between assembly and module

Module, assembly, ...

For example, compiling a *".NET Console App"*, produces single module file/assembly, with the manifest as:

```
// Metadata version: v4.0.30319
.assembly extern System.Runtime
{
  ...
}
.assembly extern System.Console
{
  ...
}
.assembly ConsoleApp
{
  ...
}
.module ConsoleApp.dll

... types/code goes here ...
```

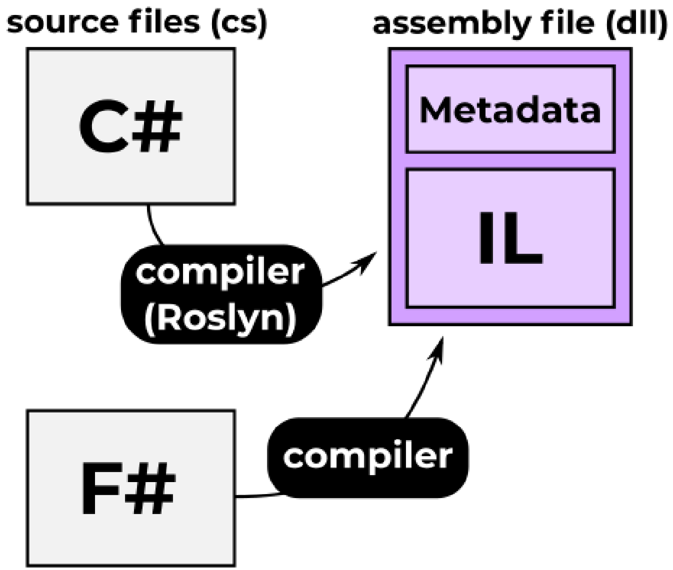
Module, assembly, ...

For example, compiling a *".NET Console App"*, produces single module file/assembly, with the manifest as:

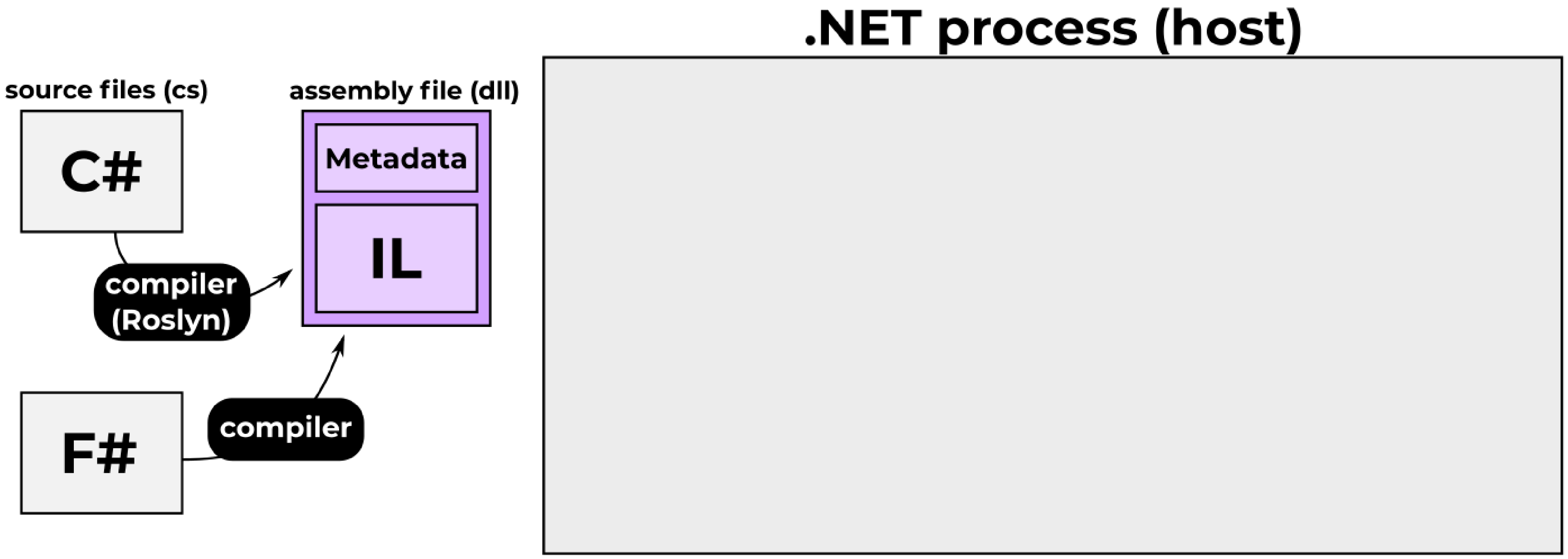
```
// Metadata version: v4.0.30319
.assembly extern System.Runtime
{
  ...
}
.assembly extern System.Console
{
  ...
}
.assembly ConsoleApp
{
  ...
}
.module ConsoleApp.dll

... types/code goes here ...
```

Sidenote: How to: Build a multifile assembly - you can build "multifile assembly" with the command line compiler (`csc.exe`) and Assembly Linker (`al.exe`), but **for .NET Framework only**.

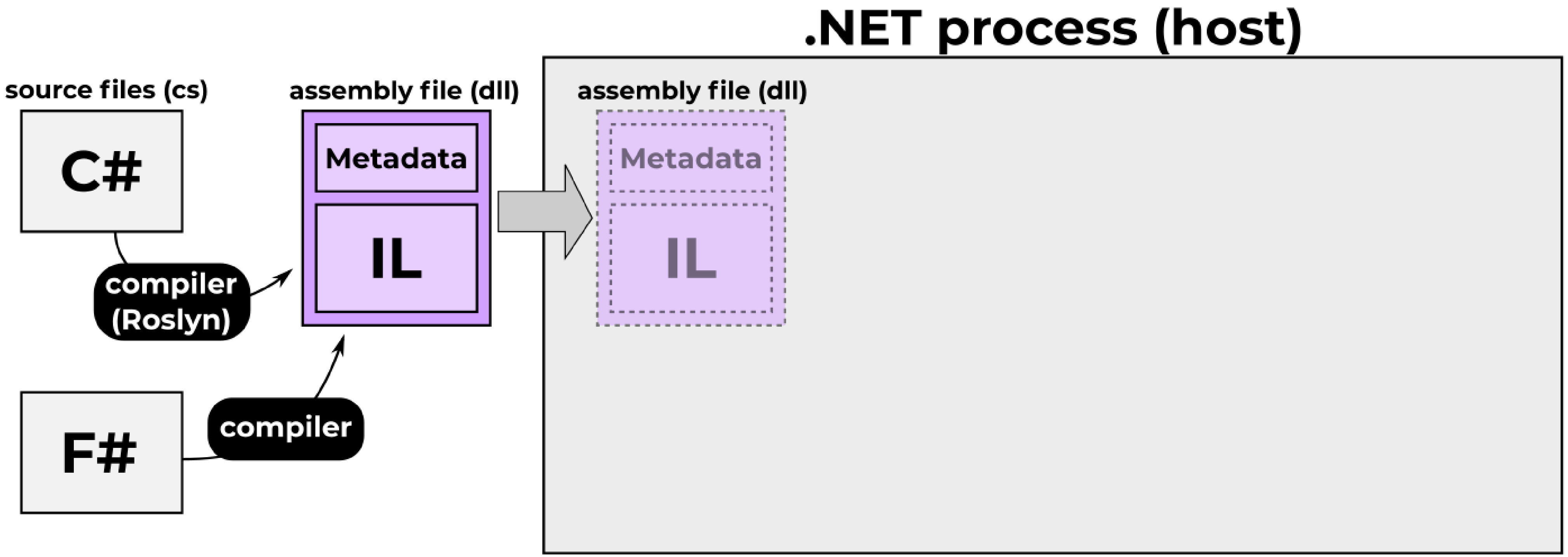
.NET ecosystem



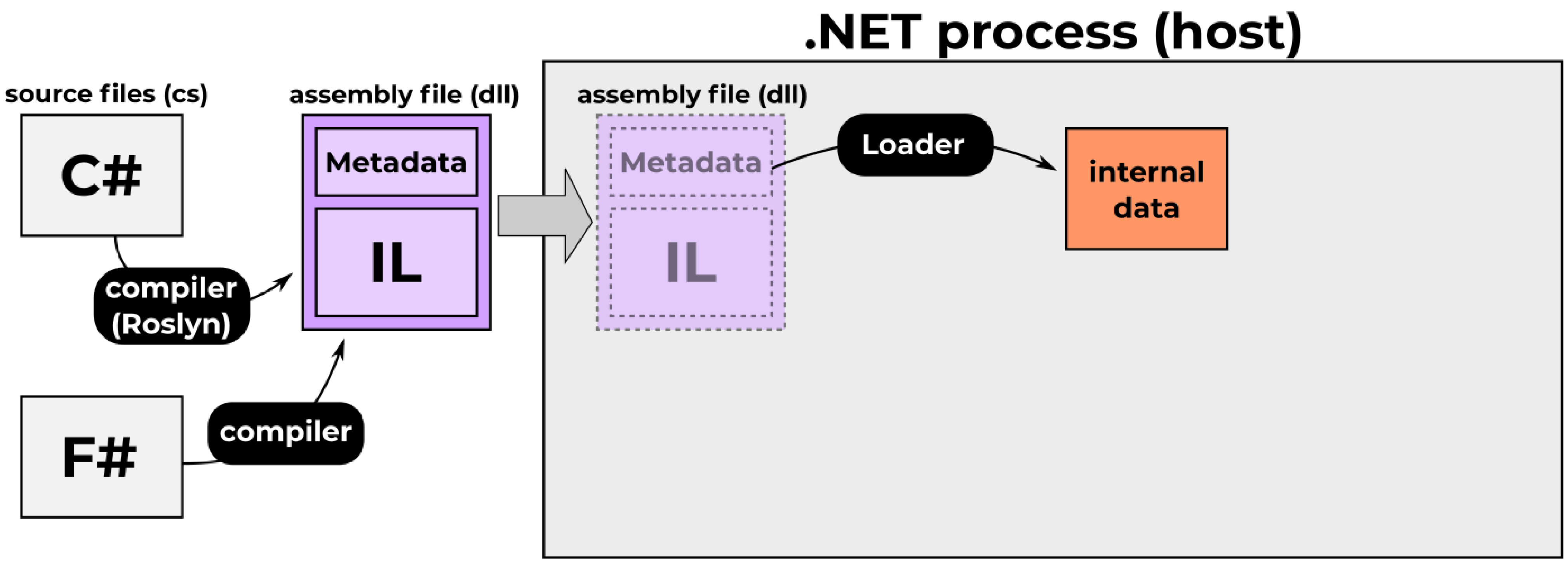
.NET ecosystem



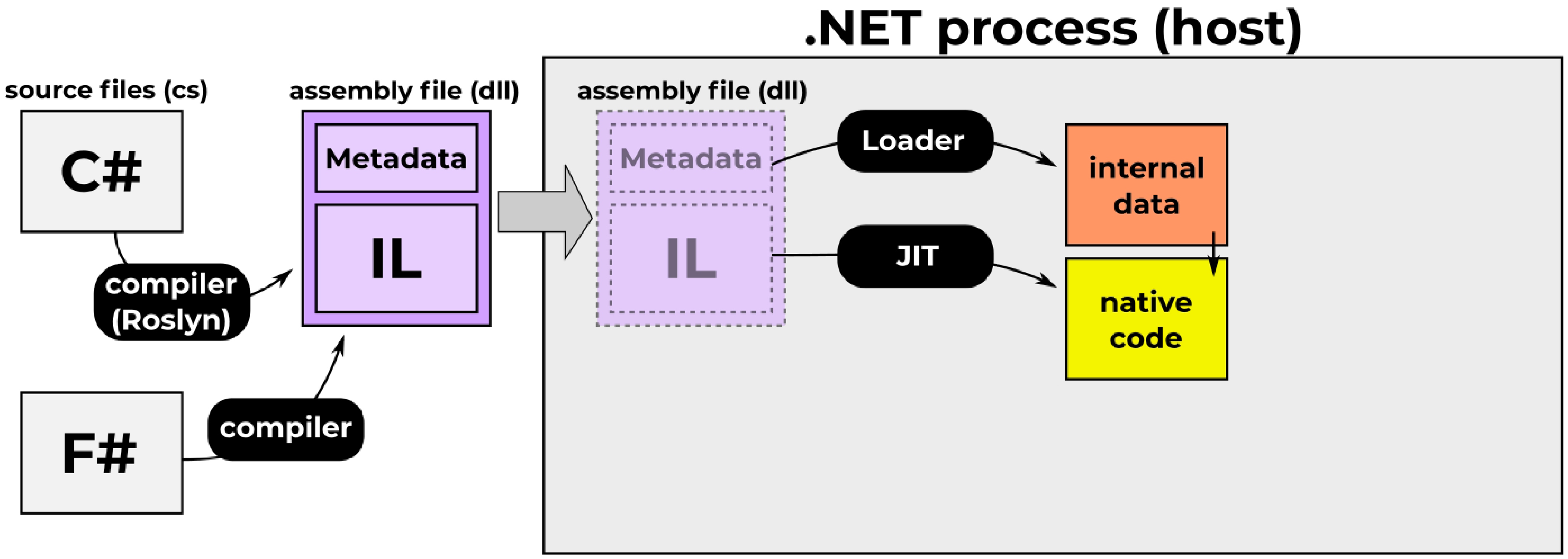
.NET ecosystem



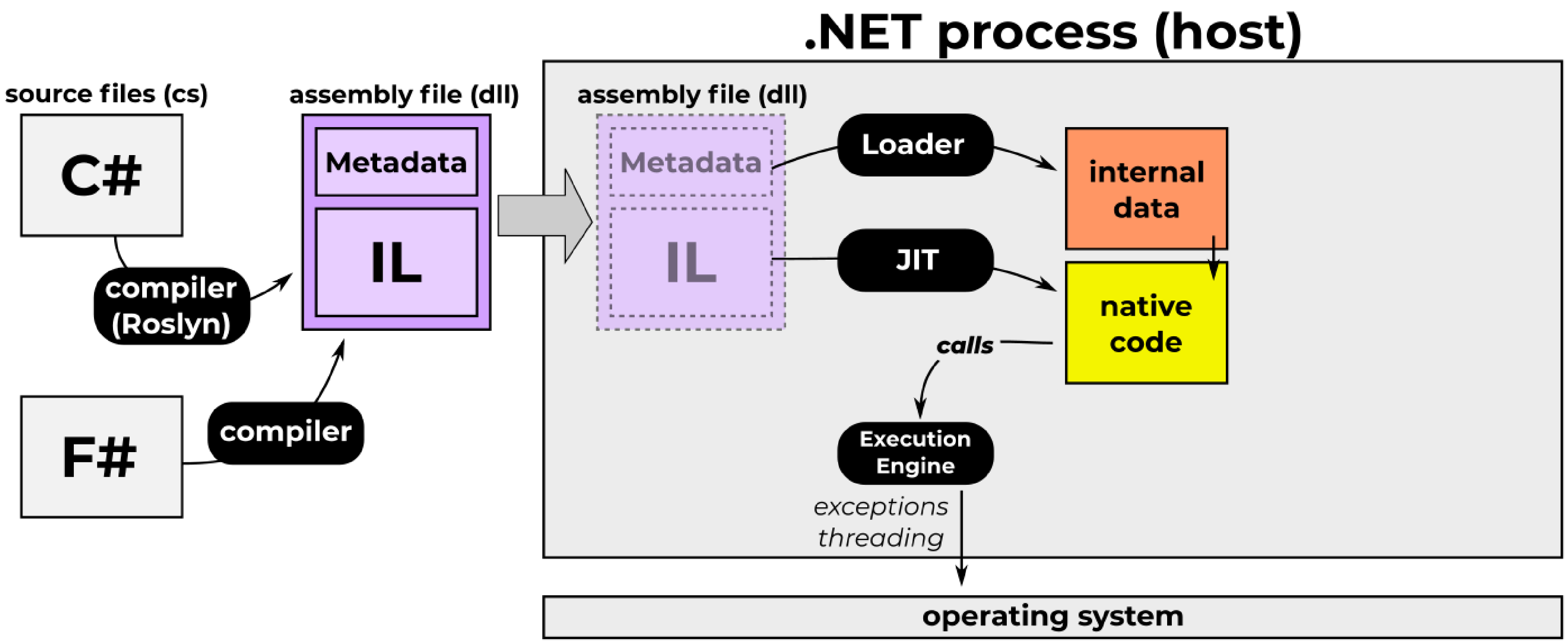
.NET ecosystem



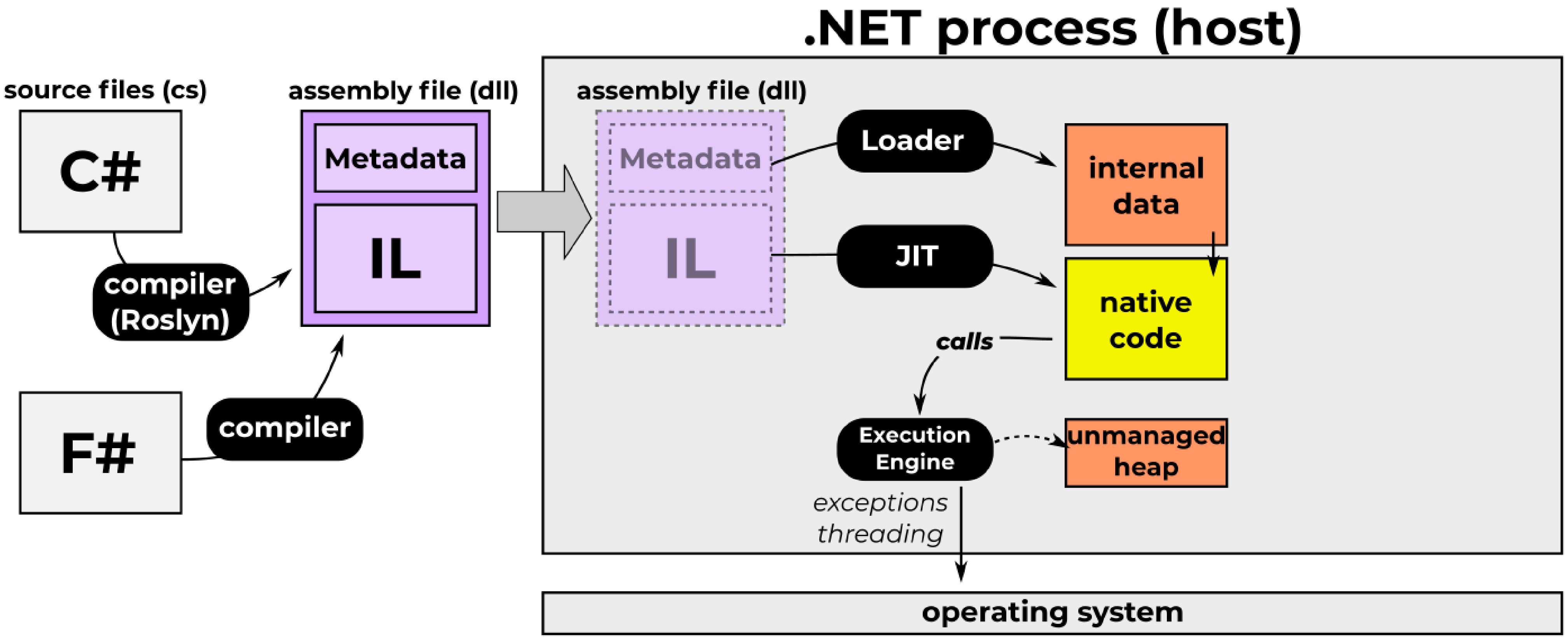
.NET ecosystem



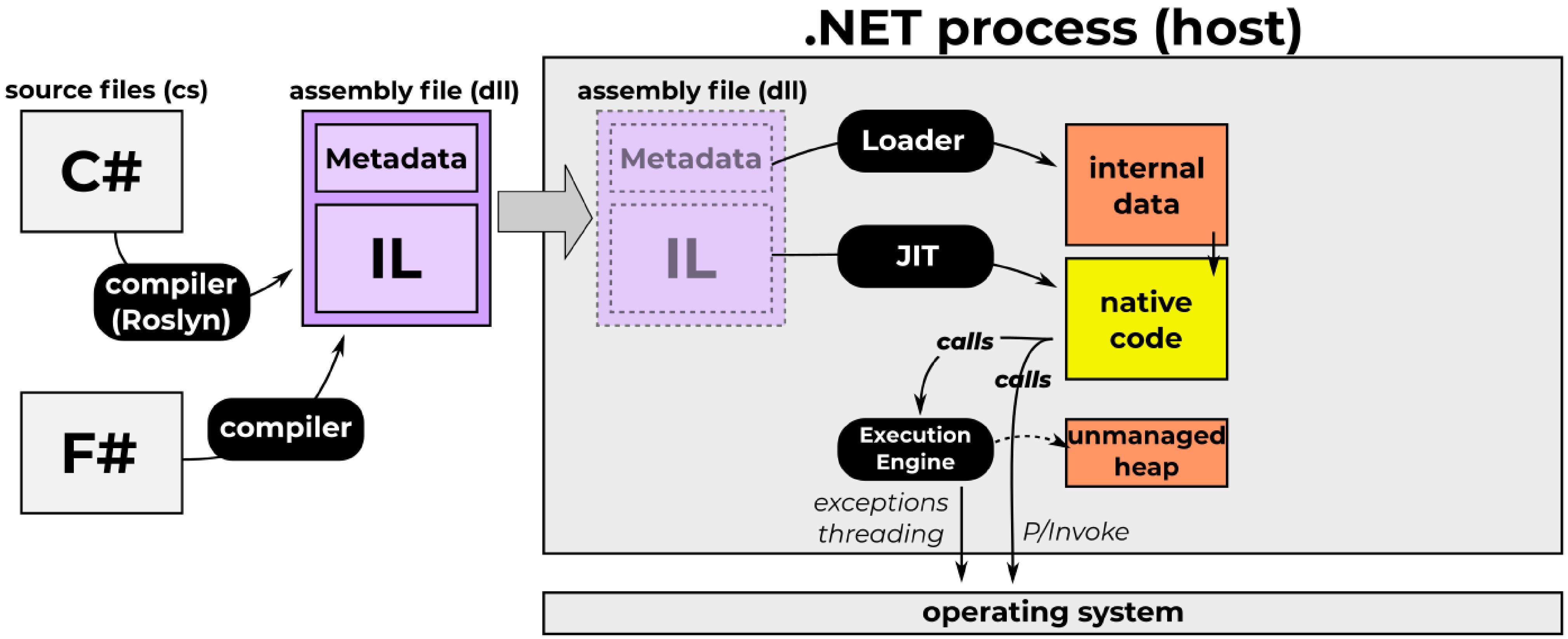
.NET ecosystem



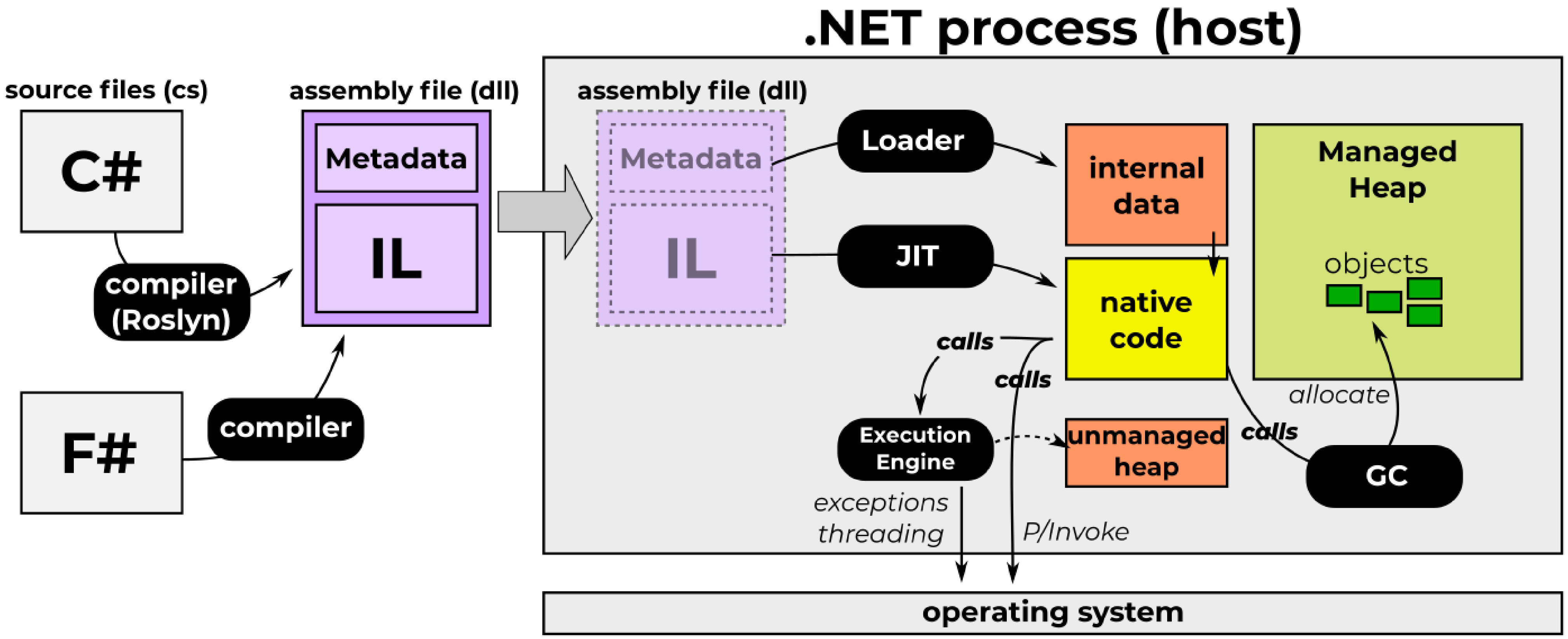
.NET ecosystem



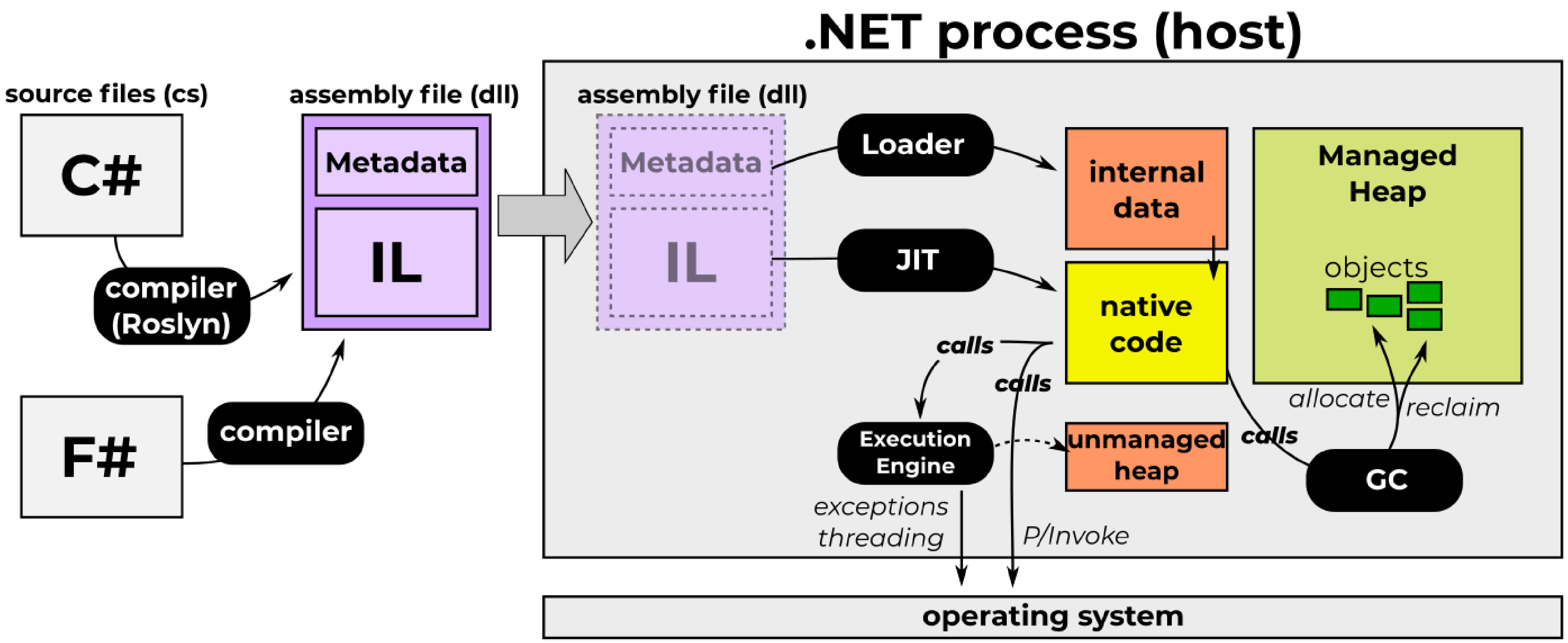
.NET ecosystem



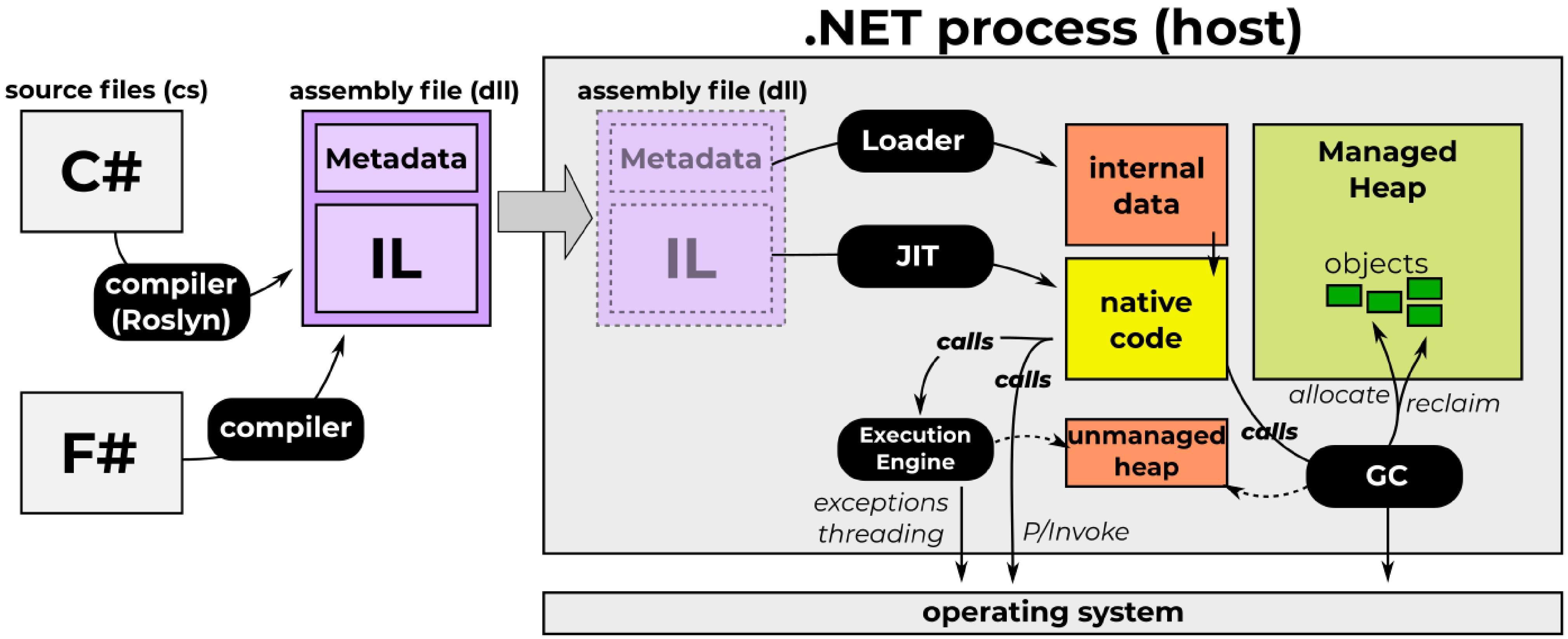
.NET ecosystem



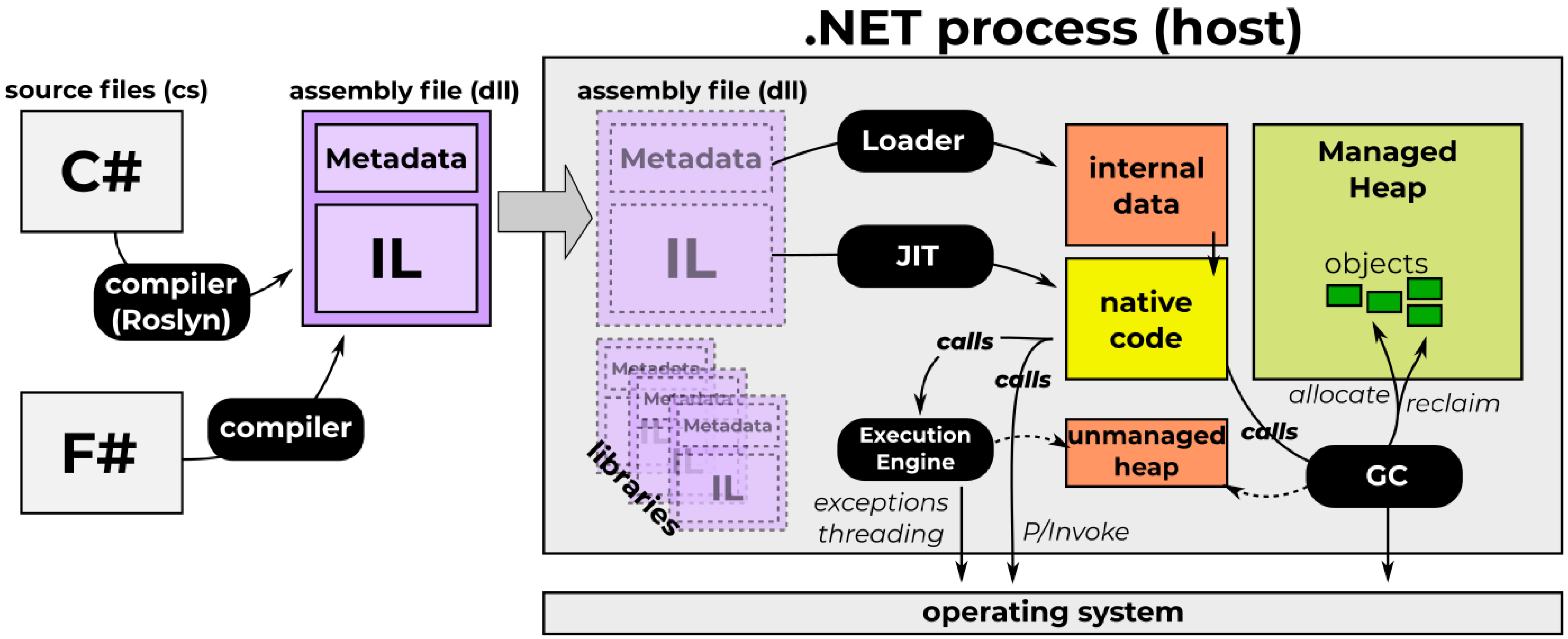
.NET ecosystem



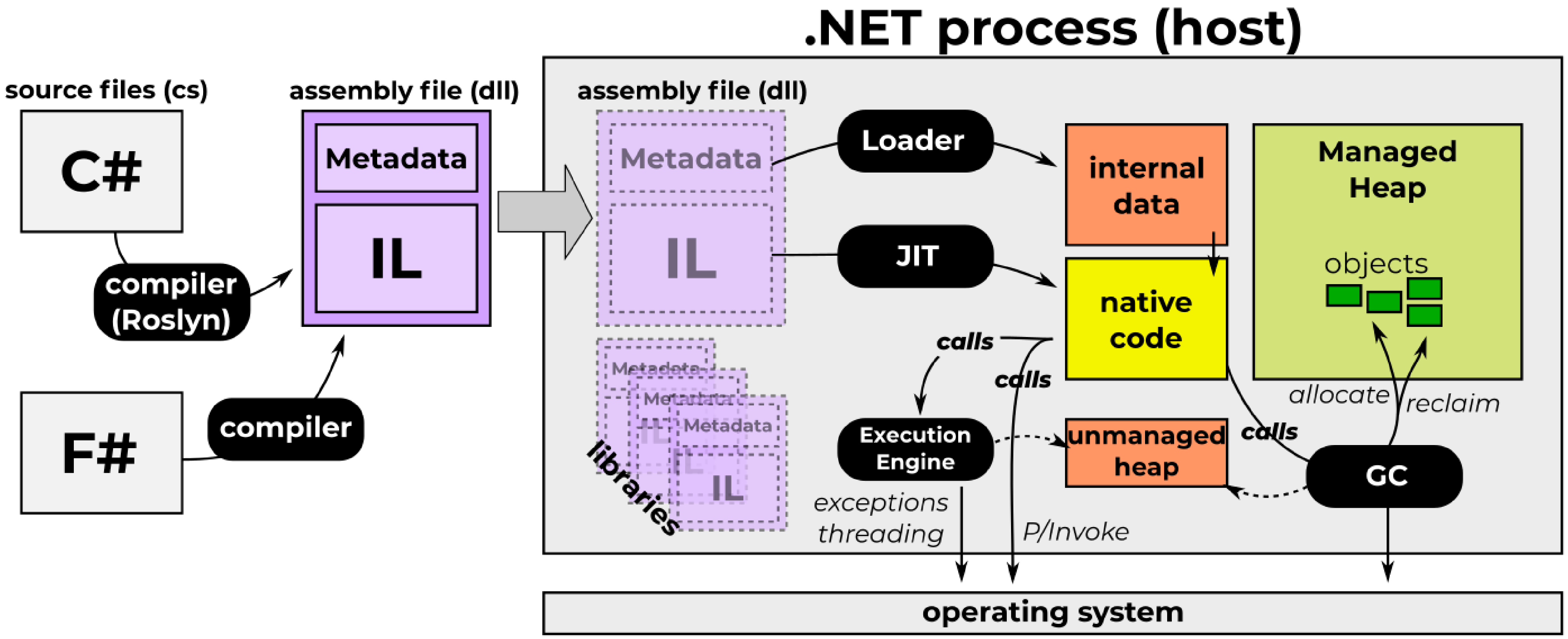
.NET ecosystem



.NET ecosystem



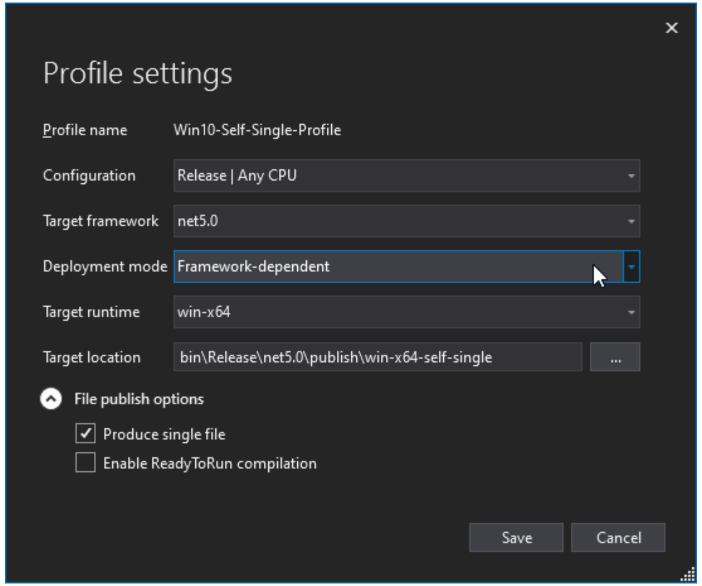
.NET ecosystem



Notice: every running .NET app loads **its own, separate .NET runtime** (and... every time it runs).

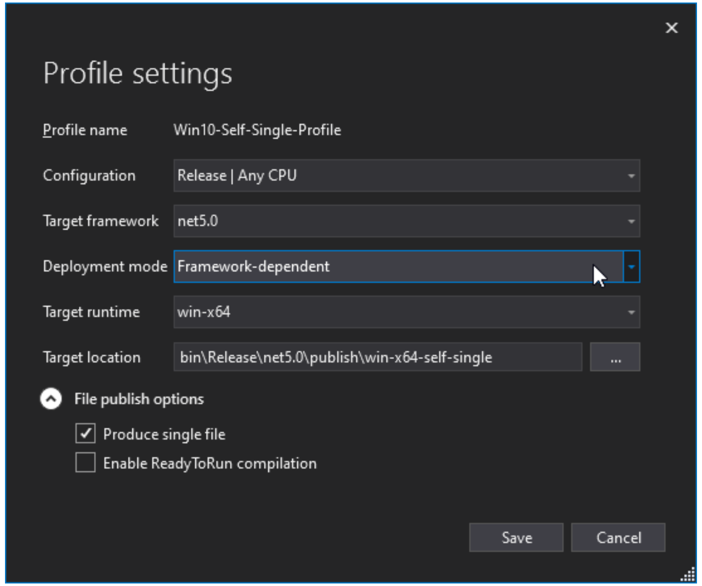
Sidenote: We will return to the topic of unloadable assemblies and `AssemblyLoadContext`

Deployment & hosting



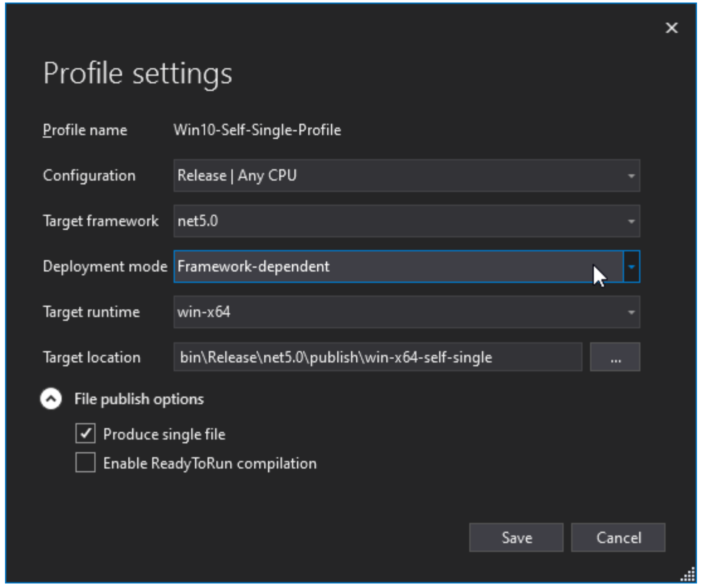
- **Target runtime** - Windows 32/64bit, Linux 64bit/ARM, MacOS 64bit, ...
- (optional) **Single file** mode - all/most dependencies (libraries, runtime) are packed into single executable file

Deployment & hosting



- Deployment mode:
  - **Framework-depentent** - it will require .NET framework/runtime installed on the target machine. Contains:
    - in normal mode:
      - assembly in PE/COFF format - the **same dll** for all OS
      - host file - exe, ELF, Mach-O...
    - in Single file mode:
      - host file with all assemblies inside - exe, ELF, Mach-O...

Deployment & hosting



- Deployment mode:
    - **Framework-depentent** - it will require .NET framework/runtime installed on the target machine. Contains:
        - in normal mode:
            - assembly in PE/COFF format - the **same dll** for all OS
            - host file - exe, ELF, Mach-O...
        - in Single file mode:
            - host file with all assemblies inside - exe, ELF, Mach-O...
    - **Self-contained** - shipping all runtime and dependencies. Contains:
        - in normal mode:
            - a LOT of files - all framework libraries, runtime files, assemblies
        - in single file mode:
            - host file with all assemblies/runtime inside - exe, ELF, Mach-O...
            - (optionally) some dependencies
- Additionally, in Single file mode for self-contained mode we can enable (experimental) **Trimming**

Deployment & hosting

win-x64

Name	Ext	Size
[..]		<DIR>
HelloLogger	dll	4,096
HelloLogger	exe	127,488
HelloLogger	pdb	9,748
HelloLogger.deps	json	478
HelloLogger.runtimeconfig	json	147

win-x64-single

Name	Ext	Size
[..]		<DIR>
HelloLogger	exe	132,394
HelloLogger	pdb	9,748

win-x64-self

Name	Ext	Size
api-ms-win-crt-multibyte..	dll	19,920
api-ms-win-crt-private-l..	dll	64,464
api-ms-win-crt-process-l..	dll	12,752
api-ms-win-crt-runtime-l..	dll	16,336
api-ms-win-crt-stdio-l1-..	dll	17,872
api-ms-win-crt-string-l1..	dll	18,384
api-ms-win-crt-time-l1-1-0	dll	14,288
api-ms-win-crt-utility-l..	dll	12,240
clrcompression	dll	748,936
clretwrc	dll	262,536
clrjit	dll	1,274,248
coreclr	dll	5,140,360
createdump	exe	55,752
dbgshim	dll	116,616
HelloLogger	dll	4,096
HelloLogger	exe	127,488
HelloLogger	pdb	9,748
HelloLogger.deps	json	32,823
HelloLogger.runtimeconfig	json	178
hostfxr	dll	315,784
hostpolicy	dll	323,968
Microsoft.CSharp	dll	993,168
Microsoft.DiaSymReader.N..	dll	1,495,800
Microsoft.VisualBasic	dll	17,272
Microsoft.VisualBasic.Core	dll	1,196,432
Microsoft.Win32.Primitives	dll	22,904
Microsoft.Win32.Registry	dll	84,344
mscordaccore	dll	1,056,632
mscordaccore_amd64_amd64..	dll	1,057,160
mscordbi	dll	1,091,976
mscorlib	dll	57,232
mscorrc	dll	142,216
netstandard	dll	114,040

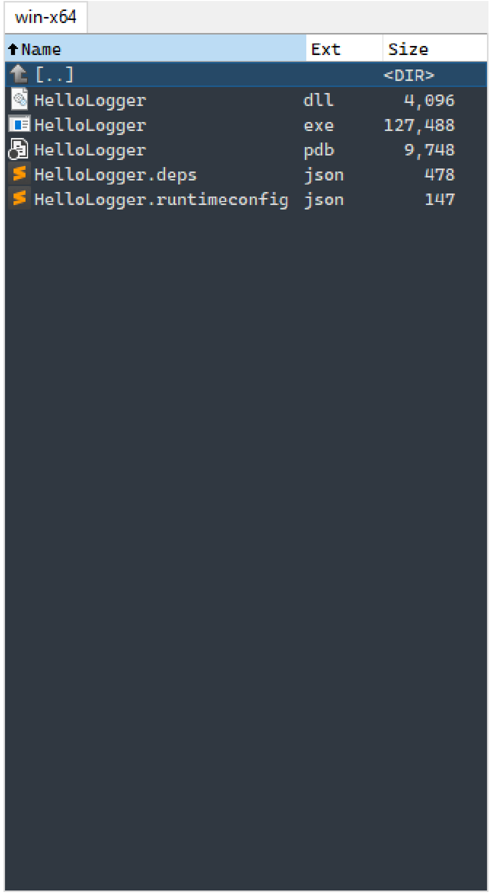
win-x64-self-single

Name	Ext	Size
[..]		<DIR>
clrcompression	dll	748,936
clrjit	dll	1,274,248
coreclr	dll	5,140,360
HelloLogger	exe	53,113,164
HelloLogger	pdb	9,748
mscordaccore	dll	1,056,632

win-x64-self-single-trim

Name	Ext	Size
[..]		<DIR>
clrcompression	dll	748,936
clrjit	dll	1,274,248
coreclr	dll	5,140,360
HelloLogger	exe	11,303,143
HelloLogger	pdb	9,516
mscordaccore	dll	1,056,632

Framework-depentent, no single file mode



- `HelloLogger.dll` - assembly containing our app (IL, metadata)
- `HelloLogger.exe` - win-x64 host
- `HelloLogger.pdb` - symbol files
- `HelloLogger.runtimeconfig.json` - as it is framework-dependent, it defines the required runtime (and its options):

```json
{
"runtimeOptions": {
  "framework": {
    "name": "Microsoft.NETCore.App",
    "version": "5.0.0"
  }
}
}
```

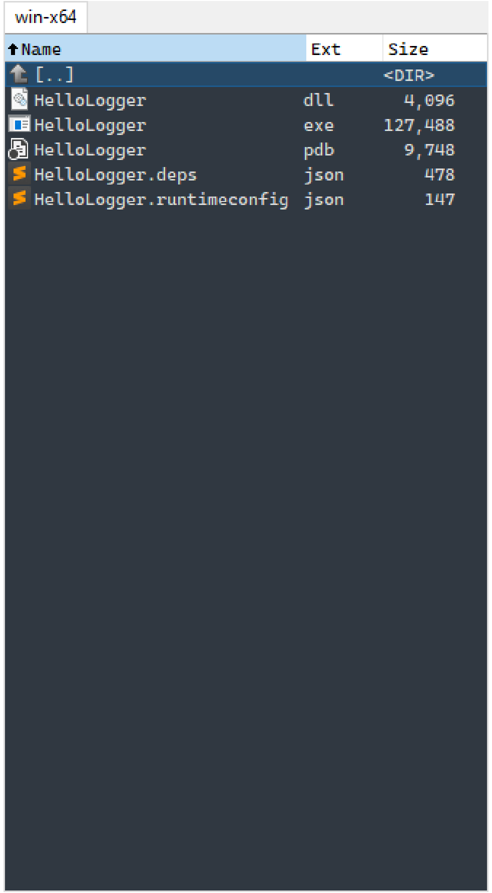
  It triggers looking for *shared runtime* version at:
  - `c:\Program Files\dotnet\shared\Microsoft.NETCore.App\` (Windows)
  - `/home/{user}/share/dotnet/shared/` (Linux)
  - `/usr/local/share/dotnet/shared/` (macOS)
- `HelloLogger.deps` - dependency manifest

Assembly anatomy

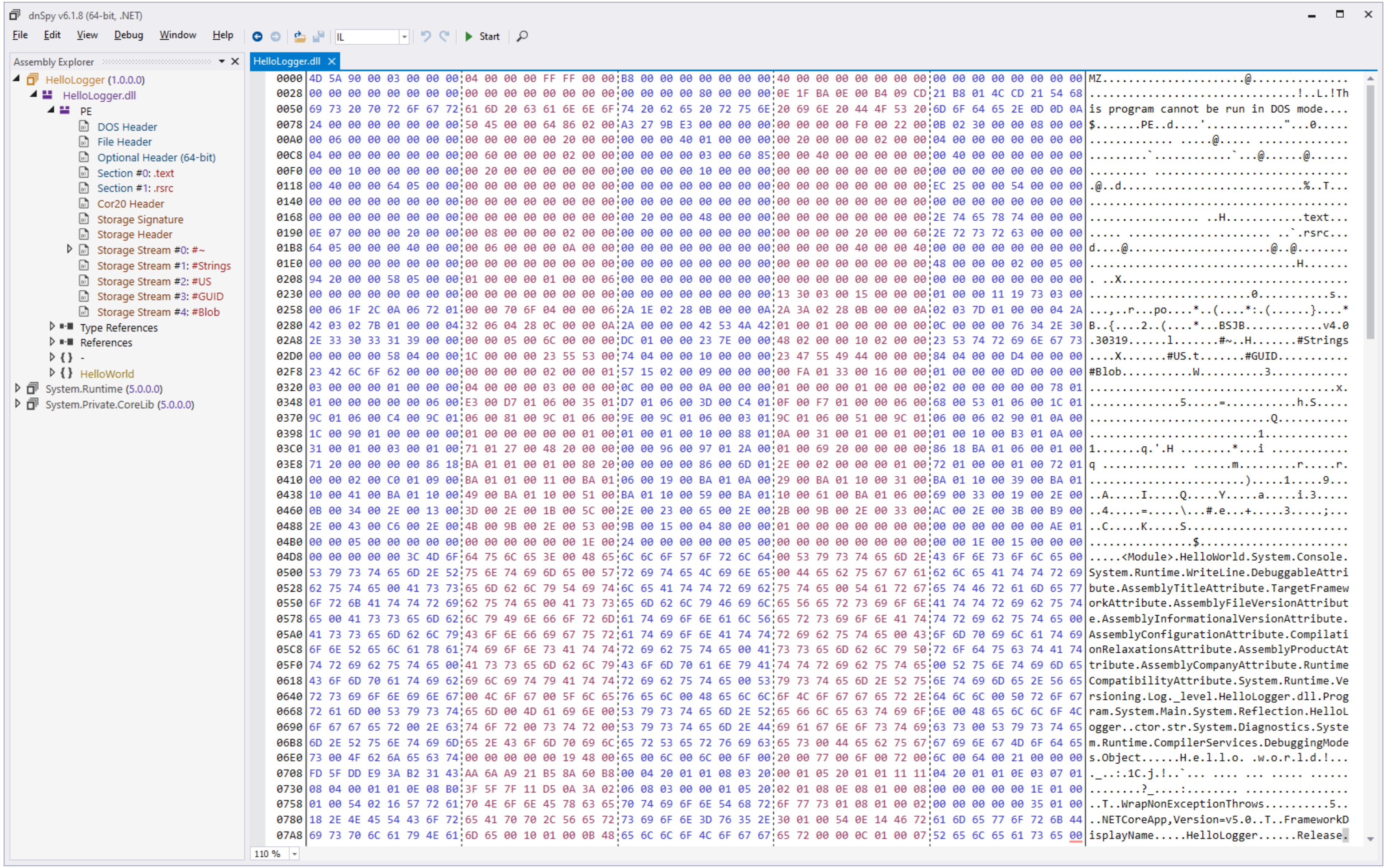
Imagine we compile such trivial C# program into *HelloLogger.dll*:

```
using System;
public class Program {
    public static void Main() {
        var logger = new Logger(3);
        var level = 44;
        logger.Log(level, "Hello world!");
    }
}
public class Logger
{
    private int _level;
    public Logger(int level) => _level = level;
    public void Log(int level, string str)
    {
        if (level >= _level) Console.WriteLine(str);
    }
}
```

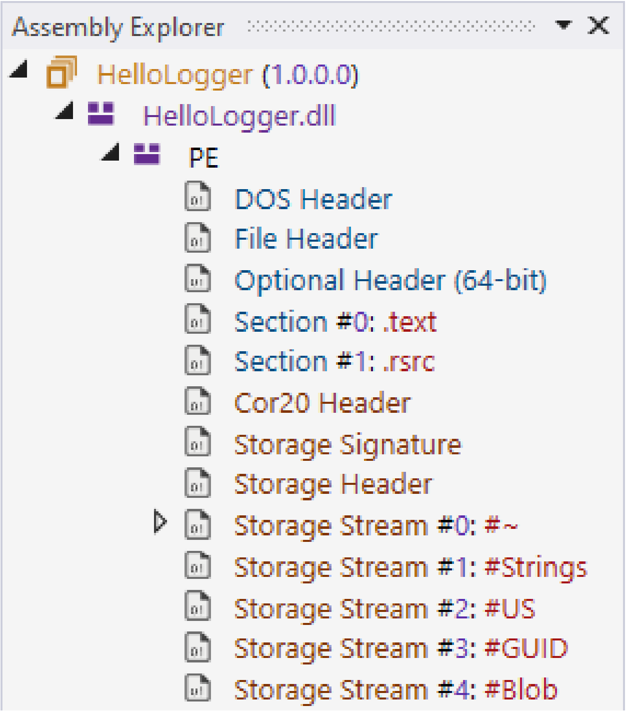
Assembly anatomy



So, we are interested in analyzing **`HelloLogger.dll`** file - assembly containing our application (types metadata, all members/methods code in CIL)


dnSpy v6.1.8 (64-bit, .NET)
File
Edit
View
Debug
Window
Help
IL
Start
Assembly Explorer
HelloLogger.dll
HelloLogger (1.0.0.0)
HelloLogger.dll
PE
DOS Header
File Header
Optional Header (64-bit)
Section #0: .text
Section #1: .rsrc
Cor20 Header
Storage Signature
Storage Header
Storage Stream #0: #~
Storage Stream #1: #Strings
Storage Stream #2: #US
Storage Stream #3: #GUID
Storage Stream #4: #Blob
Type References
References
{ } -
{ } HelloWorld
System.Runtime (5.0.0.0)
System.Private.CoreLib (5.0.0.0)
110 %

Assembly anatomy



Inside `HelloLogger.dll`:

- standard Portable Executable (PE) & Common Object File Format (COFF) headers:
  - DOS Header
  - File Header
  - Optional Header
- `.text` section - *"Executable code (free format)"* as PE documentation says - reused to **contain all the CIL code and metadata**!
  - *Cor20 header* - basic information about .NET assembly (expected runtime version, entry point)
  - storage streams
- `.rsrc` section - unmanaged resources

Assembly anatomy - storage streams

- Storage Stream #0: #~
  - Tables Stream
    - 00 Module (1)
    - 01 TypeRef (13)
    - 02 TypeDef (3)
    - 04 Field (1)
    - 06 Method (4)
    - 08 Param (3)
    - 0A MemberRef (12)
    - 0C CustomAttribute (10)
    - 11 StandAloneSig (1)
    - 20 Assembly (1)
    - 23 AssemblyRef (2)
- Storage Stream #1: #Strings
- Storage Stream #2: #US
- Storage Stream #3: #GUID
- Storage Stream #4: #Blob

- **`#~`** - all metadata about types:
  - **`Module`** - module defintion
  - **`TypeRef`** - referenced types (used by dynamic resolve)
  - **`TypeDef`** - defined types
  - **`Field`** - fields from all types
  - **`Method`** - methods froms all types
  - ...

02 TypeDef (3)

RID	Token	Offset	Flags	Name	Namespace	Extends	FieldList	MethodList	Info
1	0x02000001	0x0000039C	0	1	0	0	1	1	<Module>
2	0x02000002	0x000003AA	0x100001	0x188	0xA	0x31	1	1	HelloWorld.Program
3	0x02000003	0x000003B8	0x100001	0x1B3	0xA	0x31	1	3	HelloWorld.Logger

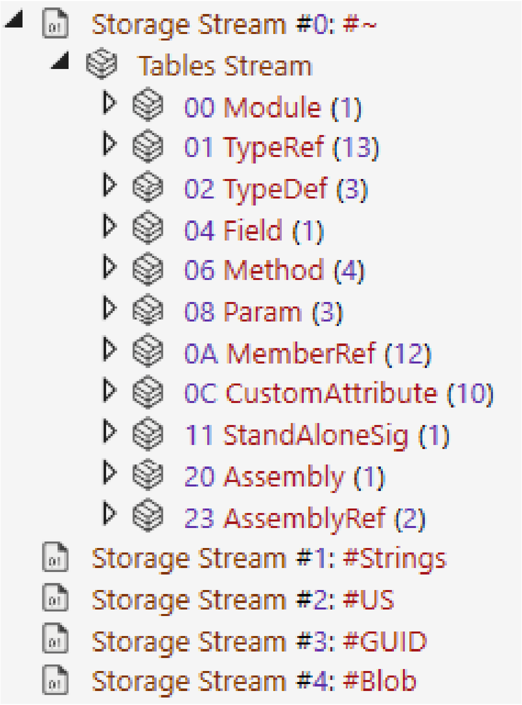
04 Field (1)

RID	Token	Offset	Flags	Name	Signature	Info
1	0x04000001	0x000003C6	1	0x171	0x27	_level

06 Method (4)

RID	Token	Offset	RVA	ImplFlags	Flags	Name	Signature	ParamList	Info
1	0x06000001	0x000003CC	0x2048	0	0x96	0x197	0x2A	1	Main
2	0x06000002	0x000003DA	0x2069	0	0x1886	0x1BA	6	1	.ctor
3	0x06000003	0x000003E8	0x2071	0	0x1886	0x1BA	1	1	.ctor
4	0x06000004	0x000003F6	0x2080	0	0x86	0x16D	0x2E	2	Log

Assembly anatomy - storage streams



- **`#Strings`** - required built-in strings (like type/method names)
- **`#US`** - user-defined strings (yes, **`"Hello world!"`**)
  (...we will return to that next weeks)
- **`#GUID`** - defined **`Guid`** values
- **`#Blob`** - all other inlined data (arrays etc.)
  (...we will return to that next weeks)

Assembly anatomy - storage streams

So, when runtime wants to execute **`Main`** method - it finds it, typically by **`Token`**, and looks for its **`RVA`** (relative virtual address - address relative to the base address of image loaded into memory):

06 Method (4)

RID	Token	Offset	RVA	ImplFlags	Flags	Name	Signature	ParamList	Info
1	0x06000001	0x000003CC	0x2048	0	0x96	0x197	0x2A	1	Main
2	0x06000002	0x000003DA	0x2069	0	0x1886	0x1BA	6	1	.ctor
3	0x06000003	0x000003E8	0x2071	0	0x1886	0x1BA	1	1	.ctor
4	0x06000004	0x000003F6	0x2080	0	0x86	0x16D	0x2E	2	Log

And then it know where the binary CIL code is (33 bytes!):

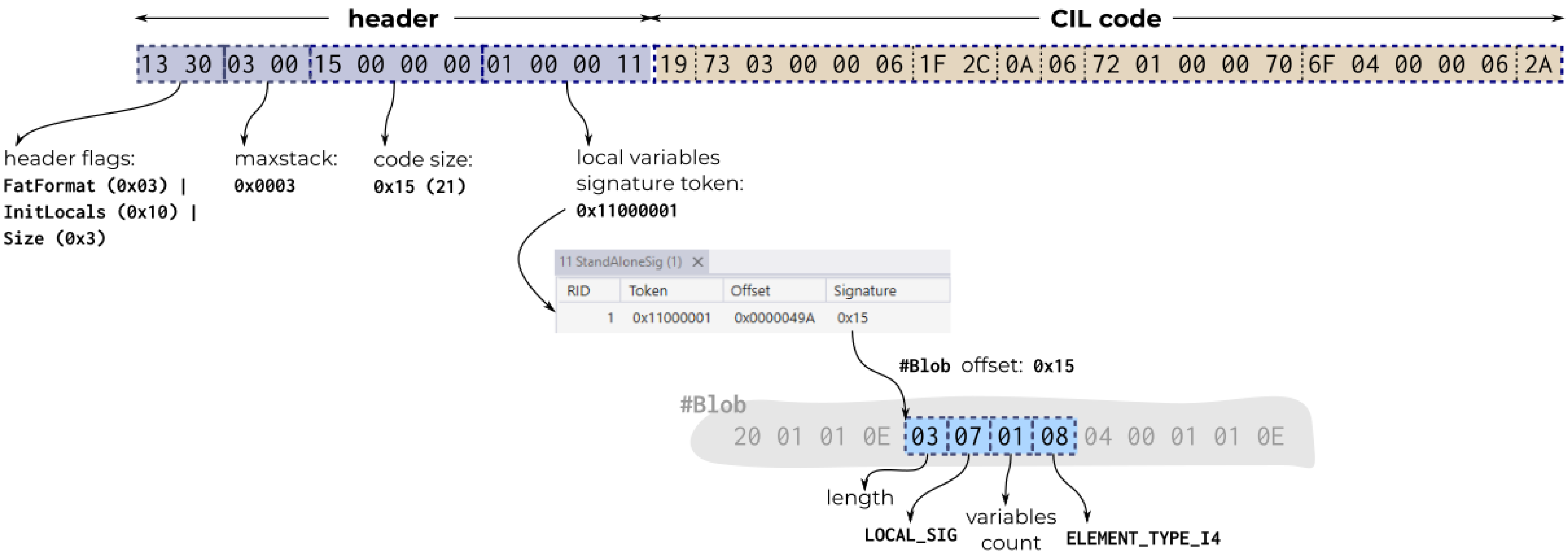
HelloLogger.dll

```
0000 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 MZ....................@.............
0027 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 ................................!..L.!
004E 54 68 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 6D 6F 64 65 2E This program cannot be run in DOS mode.
0075 0D 0D 0A 24 00 00 00 00 00 00 00 50 45 00 00 64 86 02 00 A3 27 9B E3 00 00 00 00 00 00 00 00 F0 00 22 00 0B 02 30 00 ...$.......PE..d....'..........."...0.
009C 00 08 00 00 00 06 00 00 00 00 00 00 00 00 00 00 00 20 00 00 00 00 00 40 01 00 00 00 00 20 00 00 00 02 00 00 04 00 00 ................ .....@..... .........
00C3 00 00 00 00 00 04 00 00 00 00 00 00 00 00 60 00 00 00 02 00 00 00 00 00 00 03 00 60 85 00 00 40 00 00 00 00 00 00 40 .............`...........`...@......@
00EA 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 20 00 00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 ............... ......................
0111 00 00 00 00 00 00 00 00 40 00 00 64 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ........@..d...........................
0138 EC 25 00 00 54 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .%..T..................................
015F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 48 00 00 00 00 00 00 00 00 00 .......................... ..H.........
0186 00 00 2E 74 65 78 74 00 00 00 0E 07 00 00 00 20 00 00 00 08 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 20 ...text....... ...................... 
01AD 00 00 60 2E 72 73 72 63 00 00 00 64 05 00 00 00 40 00 00 00 06 00 00 00 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..`.rsrc...d....@......................
01D4 40 00 00 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 @..@...................................
01FB 00 00 00 00 00 48 00 00 00 02 00 05 00 94 20 00 00 58 05 00 00 01 00 00 00 01 00 00 06 00 00 00 00 00 00 00 00 00 00 .....H........ ..X.....................
0222 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 13 .......................................
0249 30 03 00 15 00 00 00 01 00 00 11 19 73 03 00 00 06 1F 2C 0A 06 72 01 00 00 70 6F 04 00 00 06 2A 1E 02 28 0B 00 00 0A 0..........s.....,..r...po....*..(....
0270 2A 3A 02 28 0B 00 00 0A 02 03 7D 01 00 00 04 2A 42 03 02 7B 01 00 00 04 32 06 04 28 0C 00 00 0A 2A 00 00 00 42 53 4A *:.(......}....*B..{....2..(....*...BSJ
0297 42 01 00 01 00 00 00 00 00 0C 00 00 00 76 34 2E 30 2E 33 30 33 31 39 00 00 00 00 05 00 6C 00 00 00 DC 01 00 00 23 7E B............v4.0.30319......l.......#~
02BE 00 00 48 02 00 00 10 02 00 00 23 53 74 72 69 6E 67 73 00 00 00 00 58 04 00 00 1C 00 00 00 23 55 53 00 74 04 00 00 10 ..H.......#Strings....X.......#US.t....
02E5 00 00 00 23 47 55 49 44 00 00 00 84 04 00 00 D4 00 00 00 23 42 6C 6F 62 00 00 00 00 00 00 00 02 00 00 01 57 15 02 00 ...#GUID...........#Blob...........W...
030C 09 00 00 00 00 FA 01 33 00 16 00 00 01 00 00 00 0D 00 00 00 03 00 00 00 01 00 00 00 04 00 00 00 03 00 00 00 0C 00 00 .......3...............................
```

And as we see, its **`Signature`** & **`ParamList`** (arguments and return type) may be decoded.

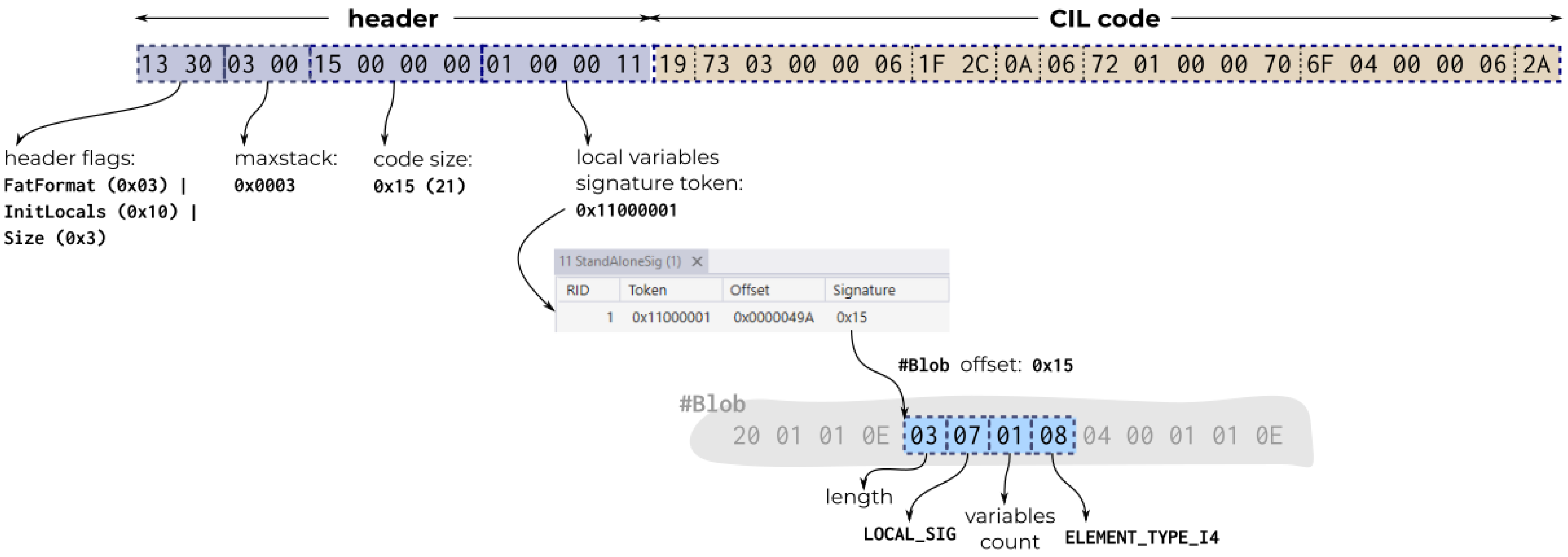
Assembly anatomy - method storage

So, in the end **`Main`** method is represented by the following structure:



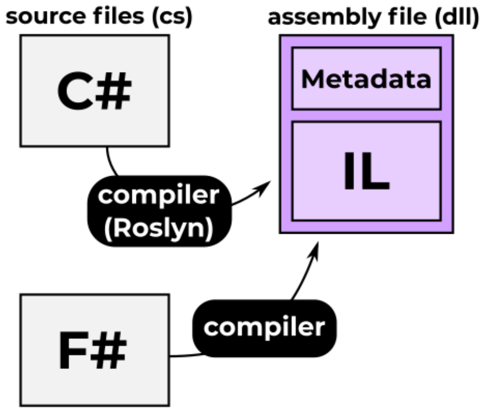
Assembly anatomy - method storage

So, in the end **`Main`** method is represented by the following structure:



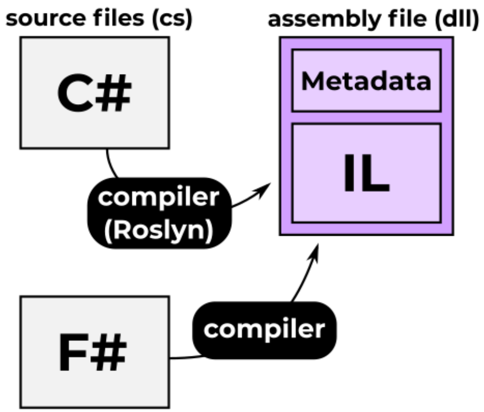
Refer to ECMA-335 22. section for more details.

Common Intermediate Language (CIL)



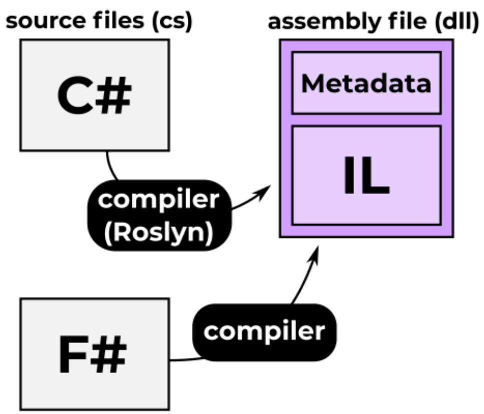
- all .NET languages are compiled to Common Intermediate Language (CIL)

Common Intermediate Language (CIL)



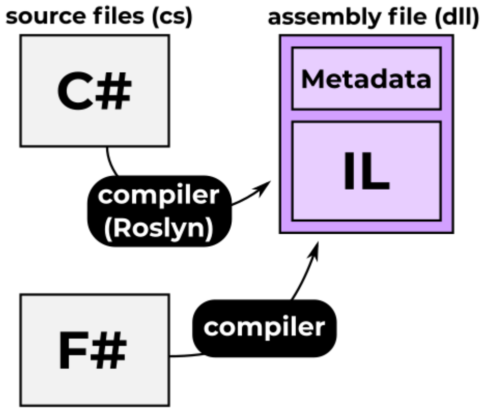
- all .NET languages are compiled to Common Intermediate Language (CIL)
  - in assembly it is stored in binary format (as we saw)

Common Intermediate Language (CIL)



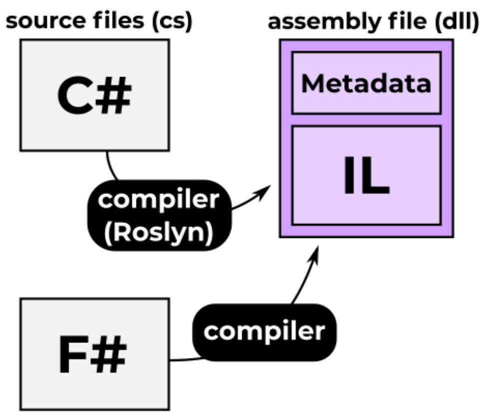
- all .NET languages are compiled to Common Intermediate Language (CIL)
  - in assembly it is stored in binary format (as we saw)
  - it may be represented also in textual format (as we will see)

Common Intermediate Language (CIL)



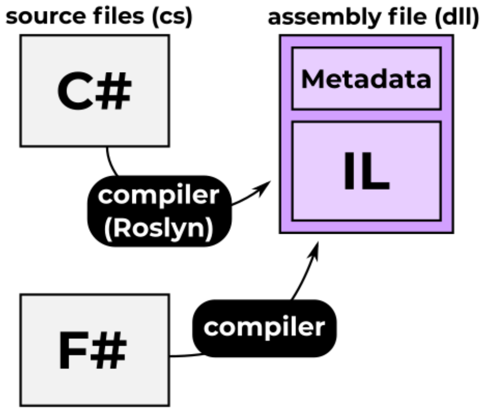
- all .NET languages are compiled to Common Intermediate Language (CIL)
  - in assembly it is stored in binary format (as we saw)
  - it may be represented also in textual format (as we will see)
  - ...exactly as in the case of native code and assembly text

Common Intermediate Language (CIL)



- all .NET languages are compiled to Common Intermediate Language (CIL)
  - in assembly it is stored in binary format (as we saw)
  - it may be represented also in textual format (as we will see)
  - …exactly as in the case of native code and assembly text
- obviously we can also *assemble* (not compile, to be strict) textual CIL into binary format (`ilasm` tool)

Common Intermediate Language (CIL)



- all .NET languages are compiled to Common Intermediate Language (CIL)
  - in assembly it is stored in binary format (as we saw)
  - it may be represented also in textual format (as we will see)
  - ...exactly as in the case of native code and assembly text
- obviously we can also *assemble* (not compile, to be strict) textual CIL into binary format (`ilasm` tool)

  (as we will do in Demo)

Common Intermediate Language (CIL)

- it consists of simple **instructions** - the whole *instruction set* is defined precisely in ECMA-335

Common Intermediate Language (CIL)

- it consists of simple **instructions** - the whole *instruction set* is defined precisely in ECMA-335
- now the tricky part...
  - it represents operations on some **abstract** (aka "virtual"), **stack-based** machine
  - and this is misleading

Common Intermediate Language (CIL)

- it consists of simple **instructions** - the whole *instruction set* is defined precisely in ECMA-335
- now the tricky part...
  - it represents operations on some **abstract** (aka "virtual"), **stack-based** machine
  - and this is misleading
- it is NOT our .NET "virtual machine" (aka Virtual Execution System, VES)
  - it is just some **abstract** machine

Common Intermediate Language (CIL)

- it consists of simple **instructions** - the whole *instruction set* is defined precisely in ECMA-335
- now the tricky part...
  - it represents operations on some **abstract** (aka "virtual"), **stack-based** machine
  - and this is misleading
- it is NOT our .NET "virtual machine" (aka Virtual Execution System, VES)
  - it is just some **abstract** machine
- it is NOT well-knowk "stack", aka thread stack
  - it IS just some **abstract** stack used by this **abstract** machine
  - it IS called **evaluation stack**

Common Intermediate Language (CIL)

- it consists of simple **instructions** - the whole *instruction set* is defined precisely in ECMA-335
- now the tricky part...
  - it represents operations on some **abstract** (aka "virtual"), **stack-based** machine
  - and this is misleading
- it is NOT our .NET "virtual machine" (aka Virtual Execution System, VES)
  - it is just some **abstract** machine
- it is NOT well-knowk "stack", aka thread stack
  - it IS just some **abstract** stack used by this **abstract** machine
  - it IS called **evaluation stack**

Summary: CIL describes **in abstract way** what your application is doing.

Common Intermediate Language (CIL)

- it consists of simple **instructions** - the whole *instruction set* is defined precisely in ECMA-335
- now the tricky part...
  - it represents operations on some **abstract** (aka "virtual"), **stack-based** machine
  - and this is misleading
- it is NOT our .NET "virtual machine" (aka Virtual Execution System, VES)
  - it is just some **abstract** machine
- it is NOT well-knowk "stack", aka thread stack
  - it IS just some **abstract** stack used by this **abstract** machine
  - it IS called **evaluation stack**

Summary: CIL describes **in abstract way** what your application is doing. Then .NET runtime uses this information to execute it - typically by Just-in-time compilation.

Common Intermediate Language (CIL)

Let's see how sample C# method is compiled into CIL:

Common Intermediate Language (CIL)

Let's see how sample C# method is compiled into CIL:

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

Common Intermediate Language (CIL)

Let's see how sample C# method is compiled into CIL:

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
.class public auto ansi beforefieldinit C
    extends [System.Private.CoreLib]System.Object
{
    .method public hidebysig
        instance int32 DoCalc (
            int32 x,
            int32 y
        ) cil managed
    {
         ldarg.1
         ldarg.2
         ble.s IL_0006

         ldarg.1
         ret

IL_0006: ldarg.2
         add
         ret
    }
}
```

Common Intermediate Language (CIL)

Let's see how sample C# method is compiled into CIL:

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
.class public auto ansi beforefieldinit C
    extends [System.Private.CoreLib]System.Object
{
    .method public hidebysig
        instance int32 DoCalc (
            int32 x,
            int32 y
        ) cil managed
    {
         ldarg.1
         ldarg.2
         ble.s IL_0006

         ldarg.1
         ret

IL_0006: ldarg.2
         add
         ret
    }
}
```

As we see it is **object-based** (classes, methods) and uses built-in types (CTS).

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1
         ldarg.2
         ble.s IL_0006
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1        /* 03 */
         ldarg.2
         ble.s IL_0006
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

ECMA-335 3.38:

"`ldarg.<length>` - load argument onto the stack

Format	Assembly Format	Description
FE 09 <*unsigned int16*>	ldarg *num*	Load argument numbered *num* onto stack.
02	ldarg.0	Load argument 0 onto stack
03	ldarg.1	Load argument 1 onto stack

***Stack Transition:* … → …, value"**

Notes:

- **`ldarg.1`** - load argument 1 (**x**) onto the *evaluation stack*
- **`DoCalc`** is an instance method (no static) - argument 0 is the class **`C`** instance reference

Evaluation stack transition: **∅** → **x**

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         ble.s IL_0006
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

ECMA-335 3.38:

"`ldarg.<length>` - load argument onto the stack

Format	Assembly Format	Description
FE 09 <*unsigned int16*>	ldarg *num*	Load argument numbered *num* onto stack.
02	ldarg.0	Load argument 0 onto stack
03	ldarg.1	Load argument 1 onto stack
04	ldarg.2	Load argument 2 onto stack

***Stack Transition:* … → …, value"**

Notes:

- **`ldarg.2`** - load argument 2 (**y**) onto the *evaluation stack*

Evaluation stack transition: **x → x, y**

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         ble.s IL_0006 /* 31 ?? */
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

ECMA-335 3.10:

"`ble.<length>` - branch on less than or equal to

Format	Assembly Format	Description
3E <**int32**>	ble *target*	branch to *target* if less than or equal to
31 <**int8**>	ble.s *target*	branch to *target* if less than or equal to, short form

Stack Transition: **…, `value1`, `value2` → … "**

Notes:

- takes out two topmost values from the evaluation stack (**`value1`**, **`value2`**)
- transfers control to *target* (instruction labelled as **`IL_0006`**) if **`value1`** ≤ **`value2`**
- **`target`** is represented as a signed offset (**`.s`** is *short form* for smaller offsets)

Evaluation stack transition: **`x`, `y`** → **∅**

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         ble.s IL_0006 /* 31 ?? */
         ldarg.1       /* 03 */
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

ECMA-335 3.38:

"`ldarg.<length>` - load argument onto the stack

Format	Assembly Format	Description
FE 09 <*unsigned int16*>	ldarg *num*	Load argument numbered *num* onto stack.
02	ldarg.0	Load argument 0 onto stack
03	ldarg.1	Load argument 1 onto stack

***Stack Transition:* … → …, value"**

Notes:

- we are in case **x** > **y**
- **`ldarg.1`** - load argument 1 (**x**) onto the *evaluation stack*

Evaluation stack transition: **ø** → **x**

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         ble.s IL_0006 /* 31 ?? */
         ldarg.1       /* 03 */
         ret           /* 2A */
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

ECMA-335 3.57:

"`ret` - return from method

Format	Assembly Format	Description
2A	Ret	Return from method, possibly returning a value

Stack Transition: **`value` (*callee*) → …, `value` (*caller*)"**

Notes:

- takes out the last value (**x**) from evaluation stack of the calee
- returns from the current method
- pushes the value on the evaluation stack of the caller

Evaluation stack transition: **x *(callee)* → …, x *(caller)***

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         ble.s IL_0006 /* 31 03 */
         ldarg.1       /* 03 */
         ret           /* 2A */
IL_0006: ldarg.1       /* 03 */
         ldarg.2
         add
         ret
  }
```

ECMA-335 3.38:

"`ldarg.<length>` - load argument onto the stack

Format	Assembly Format	Description
FE 09 <*unsigned int16*>	ldarg *num*	Load argument numbered *num* onto stack.
02	ldarg.0	Load argument 0 onto stack
03	ldarg.1	Load argument 1 onto stack

***Stack Transition:* … → …, value"**

Notes:

- we are in case **x** ≤ **y**
- **`ldarg.1`** - load argument 1 (**x**) onto the *evaluation stack*
- we've updated offset for **`ble.s`**

Evaluation stack transition: **∅** → **x**

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         ble.s IL_0006 /* 31 03 */
         ldarg.1       /* 03 */
         ret           /* 2A */
IL_0006: ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         add
         ret
  }
```

ECMA-335 3.38:

"`ldarg.<length>` - load argument onto the stack

Format	Assembly Format	Description
FE 09 <*unsigned int16*>	ldarg *num*	Load argument numbered *num* onto stack.
02	ldarg.0	Load argument 0 onto stack
03	ldarg.1	Load argument 1 onto stack
04	ldarg.2	Load argument 2 onto stack

***Stack Transition:* … → …, value"**

Notes:

- **`ldarg.2`** - load argument 2 (**y**) onto the *evaluation stack*

Evaluation stack transition: **x → x, y**

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         ble.s IL_0006 /* 31 03 */
         ldarg.1       /* 03 */
         ret           /* 2A */
IL_0006: ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         add           /* 58 */
         ret
  }
```

ECMA-335 3.1:

"add - add numeric values

Format	Assembly Format	Description
58	add	Add two values, returning a new value

Stack Transition: **..., `value1`, `value2` → ..., `result`"**

Notes:

- takes out two topmost values from the evaluation stack (**`value1`**, **`value2`**)
- pushes the result of adding them on the evaluation stack

Evaluation stack transition: **x, y → x+y**

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         ble.s IL_0006 /* 31 03 */
         ldarg.1       /* 03 */
         ret           /* 2A */
IL_0006: ldarg.1       /* 03 */
         ldarg.2       /* 04 */
         add           /* 58 */
         ret           /* 2A */
  }
```

ECMA-335 3.57:
"ret - return from method

Format	Assembly Format	Description
2A	Ret	Return from method, possibly returning a value

Stack Transition: **value (*callee*) → …, value (*caller*)"**

Notes:

- takes out the last value (**x+y**) from evaluation stack of the calee
- returns from the current method
- pushes the value on the evaluation stack of the caller

Evaluation stack transition: **x+y *(callee)* → …, x+y *(caller)***

Common Intermediate Language (CIL)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y) return x; else return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1        /* 03 */
         ldarg.2        /* 04 */
         ble.s IL_0006 /* 31 03 */
         ldarg.1        /* 03 */
         ret            /* 2A */
IL_0006: ldarg.1        /* 03 */
         ldarg.2        /* 04 */
         add            /* 58 */
         ret            /* 2A */
  }
```

Resulting CIL bytecode:

```
03 04 31 03 03 2A 03 04 58 2A
```

Common Intermediate Language (CIL)

As we see:

Common Intermediate Language (CIL)

As we see:

- a lot of push and pop operations on the evaluation stack

Common Intermediate Language (CIL)

As we see:

- a lot of push and pop operations on the evaluation stack
- some instructions have side effects, like control the target/jump (`ble`)

Common Intermediate Language (CIL)

As we see:

- a lot of push and pop operations on the evaluation stack
- some instructions have side effects, like control the target/jump (`ble`)
- some are transforming values (`add`)

Common Intermediate Language (CIL)

As we see:

- a lot of push and pop operations on the evaluation stack
- some instructions have side effects, like control the target/jump (**`ble`**)
- some are transforming values (**`add`**)

Some .NET runtime implementation could:

Common Intermediate Language (CIL)

As we see:

- a lot of push and pop operations on the evaluation stack
- some instructions have side effects, like control the target/jump (**`ble`**)
- some are transforming values (**`add`**)

Some .NET runtime implementation could:

- *interpret* CIL bytecode as-is - implement abstract evaluation stack and **execute CIL instruction one after another**
  - slow...
  - ... but has some befits (we will return to that)

Common Intermediate Language (CIL)

As we see:

- a lot of push and pop operations on the evaluation stack
- some instructions have side effects, like control the target/jump (**`ble`**)
- some are transforming values (**`add`**)

Some .NET runtime implementation could:

- *interpret* CIL bytecode as-is - implement abstract evaluation stack and **execute CIL instruction one after another**
  - slow...
  - ... but has some befits (we will return to that)
- *compile* CIL bytecode into native code - here we go **Just-in-time compiler**!
  - JIT happens at the target machine/runtime - it will produce x86/x64/ARM code for Linux/Windows/...

JIT compilation - 64-bit JIT example

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1
         ldarg.2
         ble.s IL_0006
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

JIT compilation - 64-bit JIT example

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1
         ldarg.2
         ble.s IL_0006
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

```
C.DoCalc(Int32, Int32)
       cmp edx, r8d
       jle short L0008
       mov eax, edx
       ret
L0008: lea eax, [rdx+r8]
       ret
```

So-called *calling convention* on Windows x64:

- for the first fours arguments uses **rcx**, **rdx**, **r8d** and **r9d** CPU registers
  (for the rest it uses *thread stack*)
- for the function result uses **rax** CPU register

So:

- **rcx** - (hidden) class **C** instance reference
- **rdx** - **x** argument
- **r8d** - **y** argument
- **rax** - for the result

JIT compilation - 64-bit JIT example

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1
         ldarg.2
         ble.s IL_0006
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

```
C.DoCalc(Int32, Int32)
       cmp edx, r8d
       jle short L0008
       mov eax, edx
       ret
L0008: lea eax, [rdx+r8]
       ret
```

- with the help of **cmp** instruction, compare **edx** (**x** argument) with **r8d** (**y** argument)
- if **x** ≤ **y** then jump to **L0008** instruction
- **note:** no evaluation stack at all - arguments have been translated into CPU registers

JIT compilation - 64-bit JIT example

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1
         ldarg.2
         ble.s IL_0006
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

```
C.DoCalc(Int32, Int32)
       cmp edx, r8d
       jle short L0008
       mov eax, edx
       ret
L0008: lea eax, [rdx+r8]
       ret
```

- in case of **x** > **y** - simply set the result (**eax**, shortcut of **rax**) to the **edx** (which is **x**) value
- and return

JIT compilation - 64-bit JIT example

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
.method public hidebysig
instance int32 DoCalc(int32 x, int32 y) cil managed
{
         ldarg.1
         ldarg.2
         ble.s IL_0006
         ldarg.1
         ret
IL_0006: ldarg.1
         ldarg.2
         add
         ret
  }
```

```
C.DoCalc(Int32, Int32)
       cmp edx, r8d
       jle short L0008
       mov eax, edx
       ret
L0008: lea eax, [rdx+r8]
       ret
```

- in case of **x** ≤ **y** - use a trick with **`lea`** instruction to calculate the sum of arguments (**`rdx`** and **`r8`**)...
- ...store the result in the **`eax`** register...
- and return

JIT compilation

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
C.DoCalc(Int32, Int32)
       cmp edx, r8d
       jle short L0008
       mov eax, edx
       ret
L0008: lea eax, [rdx+r8]
       ret
```

Summary:

- no evaluation stack! It's gone:
  - only CPU registers usage
  - *thread stack* will be used for local variables in more complex methods (but it is **not the same**, remember!)
- it is a pure native code, nothing special about CLR:
  - any programming language, **including C++**, could produce the same result
  - no magic calls into CLR (to check/calculate/validate/...)
  - therefore: this code runs at **native speed** 🥳
- so far, we've used optimized *Release* mode

CIL to native code (Debug)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

Debug mode is much more verbose:

- much less efficient generated code
- typically *stack frame* is created to store all local variables
- lifetime of local objects is much longer
  (we will return to that...)
- contains additional calls into CLR (fe., **`call 0x00007ffc78028010`** here is a "jit helper" **`CORINFO_HELP_DBG_IS_JUST_MY_CODE`** which checks if this is "JustMyCode" and needs to be stepped through)

CIL to native code (Debug)

```
public class C {
    public int DoCalc(int x, int y)
    {
        if (x > y)
            return x;
        else
            return x + y;
    }
}
```

```
C.DoCalc(Int32, Int32)
    push rbp
    sub rsp, 0x30
    lea rbp, [rsp+0x30]
    xor eax, eax
    mov [rbp-4], eax
    mov [rbp-8], eax
    mov [rbp+0x10], rcx
    mov [rbp+0x18], edx
    mov [rbp+0x20], r8d
    mov rax, 0x7ffc21a0c320
    cmp dword ptr [rax], 0
    je short L0031
    call 0x00007ffc78028010
L0031: nop
    mov eax, [rbp+0x18]
    cmp eax, [rbp+0x20]
    setg al
    movzx eax, al
    mov [rbp-4], eax
    cmp dword ptr [rbp-4], 0
    je short L0050
    mov eax, [rbp+0x18]
    mov [rbp-8], eax
    nop
    jmp short L005c
L0050: mov eax, [rbp+0x18]
    add eax, [rbp+0x20]
```

Debug mode is much more verbose:

- much less efficient generated code
- typically *stack frame* is created to store all local variables
- lifetime of local objects is much longer (we will return to that...)
- contains additional calls into CLR (fe., **`call 0x00007ffc78028010`** here is a "jit helper" **`CORINFO_HELP_DBG_IS_JUST_MY_CODE`** which checks if this is "JustMyCode" and needs to be stepped through)

CIL to native code (Tiered JIT)

- JIT could compile into (depending on the application mode):
  - *Release* mode - highly optimized, but it **takes time** to generate
  - *Debug* - not optimized code better for debugging, but it **is faster** generate

CIL to native code (Tiered JIT)

- JIT could compile into (depending on the application mode):
  - *Release* mode - highly optimized, but it **takes time** to generate
  - *Debug* - not optimized code better for debugging, but it **is faster** generate
- since .NET Core 2.1 **tiered compilation** allows to compile into two so-called **tiers**:
  - the first tier (*Tier0* or *QuickJIT*) - generates code more quickly, not optimized, similar to Debug version
  - the second tier (*Tier1* or "Optimizing JIT") - generates code slowly, but highly optimized, similar to Release version

CIL to native code (Tiered JIT)

- JIT could compile into (depending on the application mode):
  - *Release* mode - highly optimized, but it **takes time** to generate
  - *Debug* - not optimized code better for debugging, but it **is faster** generate
- since .NET Core 2.1 **tiered compilation** allows to compile into two so-called **tiers**:
  - the first tier (*Tier0* or *QuickJIT*) - generates code more quickly, not optimized, similar to Debug version
  - the second tier (*Tier1* or "Optimizing JIT") - generates code slowly, but highly optimized, similar to Release version
- if enabled, a method is compiled into *Tier0* and needs to be called at least 30 (*) times to be recompiled into *Tier1* (in background)

CIL to native code (Tiered JIT)

- JIT could compile into (depending on the application mode):
  - *Release* mode - highly optimized, but it **takes time** to generate
  - *Debug* - not optimized code better for debugging, but it **is faster** generate
- since .NET Core 2.1 **tiered compilation** allows to compile into two so-called **tiers**:
  - the first tier (*Tier0* or *QuickJIT*) - generates code more quickly, not optimized, similar to Debug version
  - the second tier (*Tier1* or "Optimizing JIT") - generates code slowly, but highly optimized, similar to Release version
- if enabled, a method is compiled into *Tier0* and needs to be called at least 30 (*) times to be recompiled into *Tier1* (in background)
- configuring:
  - .NET Core 2.1 and 2.2 - disabled by default
  - .NET Core 3.x and later - enabled by default
  - we can use `COMPlus_TieredCompilation` variable, `System.Runtime.TieredCompilation` runtime config or `TieredCompilation` MSBuild property

CIL to native code (Tiered JIT)

- JIT could compile into (depending on the application mode):
  - *Release* mode - highly optimized, but it **takes time** to generate
  - *Debug* - not optimized code better for debugging, but it **is faster** generate
- since .NET Core 2.1 **tiered compilation** allows to compile into two so-called **tiers**:
  - the first tier (*Tier0* or *QuickJIT*) - generates code more quickly, not optimized, similar to Debug version
  - the second tier (*Tier1* or "Optimizing JIT") - generates code slowly, but highly optimized, similar to Release version
- if enabled, a method is compiled into *Tier0* and needs to be called at least 30 (*) times to be recompiled into *Tier1* (in background)
- configuring:
  - .NET Core 2.1 and 2.2 - disabled by default
  - .NET Core 3.x and later - enabled by default
  - we can use `COMPlus_TieredCompilation` variable, `System.Runtime.TieredCompilation` runtime config or `TieredCompilation` MSBuild property
- it influences memory management! For example, tiers produces different lifetimes of local variables

(as we will see...)

CIL to native code (Tiered JIT)

- JIT could compile into (depending on the application mode):
  - *Release* mode - highly optimized, but it **takes time** to generate
  - *Debug* - not optimized code better for debugging, but it **is faster** generate
- since .NET Core 2.1 **tiered compilation** allows to compile into two so-called **tiers**:
  - the first tier (*Tier0* or *QuickJIT*) - generates code more quickly, not optimized, similar to Debug version
  - the second tier (*Tier1* or "Optimizing JIT") - generates code slowly, but highly optimized, similar to Release version
- if enabled, a method is compiled into *Tier0* and needs to be called at least 30 (*) times to be recompiled into *Tier1* (in background)
- configuring:
  - .NET Core 2.1 and 2.2 - disabled by default
  - .NET Core 3.x and later - enabled by default
  - we can use **`COMPlus_TieredCompilation`** variable, **`System.Runtime.TieredCompilation`** runtime config or **`TieredCompilation`** MSBuild property
- it influences memory management! For example, tiers produces different lifetimes of local variables (as we will see...)

(*) This is an implementation detail subject to change!

CIL to native code

And the story continues. Non-trivial methods also run on "native speed", like the following π calculation with the help of the Gregory-Leibniz series:

```
// π/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - ...
public static double GregoryLeibnizPI(int n)
{
    double sum = 0;
    for (int i = 0; i < n; i++)
    {
        double temp = 4.0 / (1 + 2 * i);
        sum += i % 2 == 0 ? temp : -temp;
    }
    return sum;
}
```

Again, there are no special calls into CLR etc.

```
GregoryLeibnizGetPI(Int32)
    L0000: vzeroupper
    L0003: vxorps xmm0, xmm0, xmm0
    L0007: xor eax, eax
    L0009: test ecx, ecx
    L000b: jle short L0052
    L000d: lea edx, [rax*2+1]
    L0014: vxorps xmm1, xmm1, xmm1
    L0018: vcvtsi2sd xmm1, xmm1, edx
    L001c: vmovsd xmm2, [C.GregoryLeibnizPI(Int32)]
    L0024: vdivsd xmm1, xmm2, xmm1
    L0028: mov edx, eax
    L002a: shr edx, 0x1f
    L002d: add edx, eax
    L002f: and edx, 0xfffffffe
    L0032: mov r8d, eax
    L0035: sub r8d, edx
    L0038: je short L0048
    L003a: vmovsd xmm2, [C.GregoryLeibnizPI(Int32)]
    L0042: vxorps xmm1, xmm1, xmm2
    L0046: jmp short L0048
    L0048: vaddsd xmm0, xmm0, xmm1
    L004c: inc eax
    L004e: cmp eax, ecx
    L0050: jl short L000d
    L0052: ret
```

CIL to native code

An important technique is **inlining**, which is a JIT power to *inline* a method body into another, instead of calling it:

```
public class C {
    public static void M() {
        var rand = new Random();
        N(rand.Next());
    }
    public static void N(int x) {
        Console.WriteLine(x);
    }
}
```

```
.method public hidebysig s
static void M () cil managed
{
    newobj instance void System.Random::.ctor()
    callvirt instance int32 System.Random::Next()
    call void C::N(int32)
    ret
}
```

CIL to native code

An important technique is **inlining**, which is a JIT power to *inline* a method body into another, instead of calling it:

```
public class C {
    public static void M() {
        var rand = new Random();
        N(rand.Next());
    }
    public static void N(int x) {
        Console.WriteLine(x);
    }
}
```

```
.method public hidebysig s
static void M () cil managed
{
    newobj instance void System.Random::.ctor()
    callvirt instance int32 System.Random::Next()
    call void C::N(int32)
    ret
}
```

So, the generated JIT code calls `Console.WriteLine` from the `M` method itself:

```
C.M()
    ...
    mov ecx, esi
    call System.Random..ctor(Int32)
    mov ecx, esi
    call System.Random.InternalSample()
    mov ecx, eax
    call System.Console.WriteLine(Int32)
    ...
```

And this happens for *simple* methods and some implementation detail-driven decisions.

CIL to native code

Unless, we disable inlining explicitly by **`MethodImpl`** attribute (which may be useful mostly in doing some fine-grained benchmarks, as we will see in the course):

```
public class C {
    public static void M() {
        var rand = new Random();
        N(rand.Next());
    }
    [MethodImpl(MethodImplOptions.NoInlining)]
    public static void N(int x) {
        Console.WriteLine(x);
    }
}
```

```
.method public hidebysig s
static void M () cil managed
{
    newobj instance void System.Random::.ctor()
    callvirt instance int32 System.Random::Next()
    call void C::N(int32)
    ret
}
```

CIL to native code

Unless, we disable inlining explicitly by **`MethodImpl`** attribute (which may be useful mostly in doing some fine-grained benchmarks, as we will see in the course):

```
public class C {
    public static void M() {
        var rand = new Random();
        N(rand.Next());
    }
    [MethodImpl(MethodImplOptions.NoInlining)]
    public static void N(int x) {
        Console.WriteLine(x);
    }
}
```

```
.method public hidebysig s
static void M () cil managed
{
    newobj instance void System.Random::.ctor()
    callvirt instance int32 System.Random::Next()
    call void C::N(int32)
    ret
}
```

So, the generated JIT code calls **`N`** from the **`M`** method, as expected:

```
C.M()
    ...
    mov ecx, esi
    call System.Random..ctor(Int32)
    mov ecx, esi
    call System.Random.InternalSample()
    mov ecx, eax
    call C.N()
    ...
```

And this happens for *simple* methods and some implementation detail-driven decisions.

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
    - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
    - there is no single paragraph/definition that summarizes it all

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
  - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
  - there is no single paragraph/definition that summarizes it all
- mostly:

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
  - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
  - there is no single paragraph/definition that summarizes it all
- mostly:
  - *"Managed data - data that is **allocated and released automatically** by the CLI, through a process called **garbage collection**."*

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
  - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
  - there is no single paragraph/definition that summarizes it all
- mostly:
  - *"Managed data - data that is **allocated and released automatically** by the CLI, through a process called **garbage collection**."*
  - *"Some implementations of the CLI will require the ability to (...) collect objects that are no longer reachable (thus providing **memory management by “garbage collection**”)."*

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
  - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
  - there is no single paragraph/definition that summarizes it all
- mostly:
  - *"Managed data - data that is **allocated and released automatically** by the CLI, through a process called **garbage collection**."*
  - *"<u>Some</u> implementations of the CLI will require the ability to (...) collect objects that are no longer reachable (thus providing **memory management by "garbage collection"**)."*
  - *"Memory store - By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a **data object**"*

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
  - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
  - there is no single paragraph/definition that summarizes it all
- mostly:
  - *"Managed data - data that is **allocated and released automatically** by the CLI, through a process called **garbage collection**."*
  - *"<u>Some</u> implementations of the CLI will require the ability to (...) collect objects that are no longer reachable (thus providing **memory management by "garbage collection"**)."*
  - *"Memory store - By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a **data object**"*
  - from *7.9.6.6 Constructors* we can read that when an object is created, *"space for the new value is **allocated in managed memory**"*

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
  - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
  - there is no single paragraph/definition that summarizes it all
- mostly:
  - *"Managed data - data that is **allocated and released automatically** by the CLI, through a process called **garbage collection**."*
  - *"<u>Some</u> implementations of the CLI will require the ability to (…) collect objects that are no longer reachable (thus providing **memory management by "garbage collection"**)."*
  - *"Memory store - By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a **data object**"*
  - from *7.9.6.6 Constructors* we can read that when an object is created, *"space for the new value is **allocated in managed memory**"*
- summarizing (pseudo-definition):

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
    - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
    - there is no single paragraph/definition that summarizes it all
- mostly:
    - *"Managed data - data that is **allocated and released automatically** by the CLI, through a process called **garbage collection**."*
    - *"Some implementations of the CLI will require the ability to (...) collect objects that are no longer reachable (thus providing **memory management by "garbage collection"**)."*
    - *"Memory store - By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a **data object**"*
    - from *7.9.6.6 Constructors* we can read that when an object is created, *"space for the new value is **allocated in managed memory**"*
- summarizing (pseudo-definition):
    - .NET uses **managed memory** (aka **Managed Heap**) where **managed objects** are **allocated** (and optionally, automatically **released**) by **garbage collection**

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
  - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
  - there is no single paragraph/definition that summarizes it all
- mostly:
  - *"Managed data - data that is **allocated and released automatically** by the CLI, through a process called **garbage collection**."*
  - *"<u>Some</u> implementations of the CLI will require the ability to (...) collect objects that are no longer reachable (thus providing **memory management by "garbage collection"**)."*
  - *"Memory store - By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a **data object**"*
  - from *7.9.6.6 Constructors* we can read that when an object is created, *"space for the new value is **allocated in managed memory**"*
- summarizing (pseudo-definition):
  - .NET uses **managed memory** (aka **Managed Heap**) where **managed objects** are **allocated** (and optionally, automatically **released**) by **garbage collection**
  - there are no enforced garbage collection implementation details in ECMA

CLI, CIL and memory management

- where's the GC so far? We haven't seen anything related to memory management, yet.
- in ECMA-335 the topic of memory management is little scattered and vague:
  - *"managed data"*, *"memory store"* and even *"garbage collection"* mentioned here and there
  - there is no single paragraph/definition that summarizes it all
- mostly:
  - *"Managed data - data that is **allocated and released automatically** by the CLI, through a process called **garbage collection**."*
  - *"Some implementations of the CLI will require the ability to (...) collect objects that are no longer reachable (thus providing **memory management by "garbage collection"**)."*
  - *"Memory store - By "memory store" we mean the regular process memory that the CLI operates within. Conceptually, this store is simply an array of bytes. The index into this array is the address of a **data object**"*
  - from *7.9.6.6 Constructors* we can read that when an object is created, *"space for the new value is **allocated in managed memory**"*
- summarizing (pseudo-definition):
  - .NET uses **managed memory** (aka **Managed Heap**) where **managed objects** are **allocated** (and optionally, automatically **released**) by **garbage collection**
  - there are no enforced garbage collection implementation details in ECMA
  - only some restrictions to allow, e.g. compacting/moving GC etc.

CLI, CIL and memory management

- putting aside a little vague definitions, in the end we have two CIL instructions for **allocating** new objects:
  - `newobj` (ECMA-335 4.20) - creates a new object or a new instance of a value type
  - `newarr` (ECMA-335 4.19) - a new zero-based, one-dimensional array

CLI, CIL and memory management

- putting aside a little vague definitions, in the end we have two CIL instructions for **allocating** new objects:
  - `newobj` (ECMA-335 4.20) - creates a new object or a new instance of a value type
  - `newarr` (ECMA-335 4.19) - a new zero-based, one-dimensional array
- those two are the only ones about "memory management":

CLI, CIL and memory management

- putting aside a little vague definitions, in the end we have two CIL instructions for **allocating** new objects:
  - `newobj` (ECMA-335 4.20) - creates a new object or a new instance of a value type
  - `newarr` (ECMA-335 4.19) - a new zero-based, one-dimensional array
- those two are the only ones about "memory management":
  - there is no `delobj` or `delarr` to reclaim memory

CLI, CIL and memory management

- putting aside a little vague definitions, in the end we have two CIL instructions for **allocating** new objects:
  - **`newobj`** (ECMA-335 4.20) - creates a new object or a new instance of a value type
  - **`newarr`** (ECMA-335 4.19) - a new zero-based, one-dimensional array
- those two are the only ones about "memory management":
  - there is no **`delobj`** or **`delarr`** to reclaim memory
  - memory reclamation is automatic, after it is discovered that object is no longer reachable

    (how it is being discovered will be covered in next modules)

CLI, CIL and memory management

```
using System;
public class Program {
    public static void Main() {
        var logger = new Logger(3);
        var level = 44;
        logger.Log(level, "Hello world!");
    }
}
public class Logger
{
    private int _level;
    public Logger(int level) => _level = level;
    public void Log(int level, string str)
    {
        if (level >= _level) Console.WriteLine(str);
    }
}
```

`Main` was represented in CIL:

```
19 73 03 00 00 06 1F 2C 0A 06 72 01 00 00 70 6F
04 00 00 06 2A
```

CLI, CIL and memory management

```
using System;
public class Program {
    public static void Main() {
        var logger = new Logger(3);
        var level = 44;
        logger.Log(level, "Hello world!");
    }
}
public class Logger
{
    private int _level;
    public Logger(int level) => _level = level;
    public void Log(int level, string str)
    {
        if (level >= _level) Console.WriteLine(str);
    }
}
```

`Main` was represented in CIL:

`19 73 03 00 00 06 1F 2C 0A 06 72 01 00 00 70 6F 04 00 00 06 2A`

Which is in textual form:

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint
  .locals init ([0] int32 level)

  ldc.i4.3                  /* 19         */
  newobj   instance void HelloWorld.Logger::.ctor(...
                            /* 7303000006 */
  ldc.i4.s 44               /* 1F2C       */
  stloc.0                   /* 0A         */
  ldloc.0                   /* 06         */
  ldstr    "Hello world!"      /* 7201000070 */
  callvirt instance void HelloWorld.Logger::Log(...
                            /* 6F04000006 */
  ret                       /* 2A         */
}
```

CLI, CIL and memory management

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint
  .locals init ([0] int32 level)

  ldc.i4.3                   /* 19         */
  newobj   instance void HelloWorld.Logger::.ctor(int3
                             /* 7303000006 */
  ldc.i4.s 44                /* 1F2C       */
  stloc.0                    /* 0A         */
  ldloc.0                    /* 06         */
  ldstr    "Hello world!"       /* 7201000070 */
  callvirt instance void HelloWorld.Logger::Log(int32,
                             /* 6F04000006 */
  ret                        /* 2A         */
}
```

Sidenote: We see some redundancy here produced by C# compiler:

- **`stloc.0`** - pop value from evaluation stack to local variable 0
- **`ldloc.0`** - load local variable 0 onto the evaluation stack

Which is no-op and it doesn't make sense 😇

CLI, CIL and memory management

Sidenote (cont.): Mentally, we can get rid of those two instructions!

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint
  .locals init ([0] int32 level)

  ldc.i4.3
  newobj   instance void HelloWorld.Logger::.ctor(int32)
  ldc.i4.s 44
  stloc.0
  ldloc.0
  ldstr    "Hello world!"
  callvirt instance void HelloWorld.Logger::Log(int32, string)
  ret
}
```

CLI, CIL and memory management

Sidenote (cont.): Mentally, we can get rid of those two instructions!

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint
  .locals init ([0] int32 level)

  ldc.i4.3
  newobj   instance void HelloWorld.Logger::.ctor(int32)
  ldc.i4.s 44
  ldstr    "Hello world!"
  callvirt instance void HelloWorld.Logger::Log(int32, string)
  ret
}
```

CLI, CIL and memory management

Sidenote (cont.): And now, we don't need local variables:

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint
  .locals init ([0] int32 level)

  ldc.i4.3
  newobj   instance void HelloWorld.Logger::.ctor(int32)
  ldc.i4.s 44
  ldstr    "Hello world!"
  callvirt instance void HelloWorld.Logger::Log(int32, string)
  ret
}
```

CLI, CIL and memory management

Sidenote (cont.): And now, we don't need local variables:

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint

  ldc.i4.3
  newobj   instance void HelloWorld.Logger::.ctor(int32)
  ldc.i4.s 44
  ldstr    "Hello world!"
  callvirt instance void HelloWorld.Logger::Log(int32, string)
  ret
}
```

CIL, memory management and JIT

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint
  .locals init ([0] int32 level)

  ldc.i4.3
  newobj   instance void HelloWorld.Logger::.ctor(...
  ldc.i4.s 44
  ldstr    "Hello world!"
  callvirt instance void HelloWorld.Logger::Log(...
  ret
}
```

```
Program.Main()
    sub rsp, 0x28
    mov rcx, 0x7ffc2265d050
    call 0x00007ffc77ee9b70
    mov dword ptr [rax+8], 3
    mov r8, 0x257c846d0c0
    mov r8, [r8]
    mov rcx, rax
    mov edx, 0x2c
    add rsp, 0x28
    jmp Logger.Log(Int32, System.String)
```

CIL, memory management and JIT

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint
  .locals init ([0] int32 level)

  ldc.i4.3
  newobj   instance void HelloWorld.Logger::.ctor(...
  ldc.i4.s 44
  ldstr    "Hello world!"
  callvirt instance void HelloWorld.Logger::Log(...
  ret
}
```

```
Program.Main()
    sub rsp, 0x28
    mov rcx, 0x7ffc2265d050 (MT: HelloWorld.Logger)
    call 0x00007ffc77ee9b70 (JIT_TrialAllocSFastMP...
    mov dword ptr [rax+8], 3
    mov r8, 0x257c846d0c0
    mov r8, [r8]
    mov rcx, rax
    mov edx, 0x2c
    add rsp, 0x28
    jmp Logger.Log(Int32, System.String)
```

CIL, memory management and JIT

```
.method public hidebysig static
void Main () cil managed
{
  .maxstack 3
  .entrypoint
  .locals init ([0] int32 level)

  ldc.i4.3
  newobj   instance void HelloWorld.Logger::.ctor(...
  ldc.i4.s 44
  ldstr    "Hello world!"
  callvirt instance void HelloWorld.Logger::Log(...
  ret
}
```

```
Program.Main()
    sub rsp, 0x28
    mov rcx, 0x7ffc2265d050 (MT: HelloWorld.Logger)
    call 0x00007ffc77ee9b70 (JIT_TrialAllocSFastMP...
    mov dword ptr [rax+8], 3
    mov r8, 0x257c846d0c0
    mov r8, [r8]
    mov rcx, rax
    mov edx, 0x2c
    add rsp, 0x28
    jmp Logger.Log(Int32, System.String)
```

- again, we don't see anything about evaluation stack - only CPU registers usage
- `new` expression from C# has been translated to `newobj` and constructor call
- `newobj` is JITted to a call into CLR **allocator**
  - now our code is indeed "managed" - it uses CLR :)
- constructor call has been **inlined**
- there is nothing about garbage collection
  - runtime will discover that `Logger` is not needed - we'll cover that!

CLI, CIL and memory management

Sidenote: there is also `localloc` CIL instruction, exposed to C# as `stackalloc` expression, which allocates bytes from so-called **local dynamic memory pool**. We'll cover that much later as more advanced use case.

Is is worth to know CIL?

- you DON'T need it to:

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)
    (as we will see in the demo)

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)

    (as we will see in the demo)
- you MAY need it to:

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)

    (as we will see in the demo)
- you MAY need it to:
  - understand some machine generated code (P/Invoke stubs) or dumps

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)
    (as we will see in the demo)
- you MAY need it to:
  - understand some machine generated code (P/Invoke stubs) or dumps
  - write specialized code using IL instructions not emitted by C#/F# compiler:
    - like **`tail`** - but... JIT does it even without it
    - like **`calli`** and **`ldftn`** - but... in C# 9 there are exposed as *function pointers*
    - use **`Reflection.Emit`** for the above - Mobius example

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)

    (as we will see in the demo)
- you MAY need it to:
  - understand some machine generated code (P/Invoke stubs) or dumps
  - write specialized code using IL instructions not emitted by C#/F# compiler:
    - like **`tail`** - but... JIT does it even without it
    - like **`calli`** and **`ldftn`** - but... in C# 9 there are exposed as *function pointers*
    - use **`Reflection.Emit`** for the above - Mobius example
  - write code omitting default type safety
    - like **`Unsafe`** class

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)
    (as we will see in the demo)
- you MAY need it to:
  - understand some machine generated code (P/Invoke stubs) or dumps
  - write specialized code using IL instructions not emitted by C#/F# compiler:
    - like `tail` - but... JIT does it even without it
    - like `calli` and `ldftn` - but... in C# 9 there are exposed as *function pointers*
    - use `Reflection.Emit` for the above - Mobius example
  - write code omitting default type safety
    - like `Unsafe` class
  - write code weaving/IL rewritting - like Fody or any custom profiling/monitoring tool

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)
    (as we will see in the demo)
- you MAY need it to:
  - understand some machine generated code (P/Invoke stubs) or dumps
  - write specialized code using IL instructions not emitted by C#/F# compiler:
    - like **`tail`** - but... JIT does it even without it
    - like **`calli`** and **`ldftn`** - but... in C# 9 there are exposed as *function pointers*
    - use **`Reflection.Emit`** for the above - Mobius example
  - write code omitting default type safety
    - like **`Unsafe`** class
  - write code weaving/IL rewritting - like Fody or any custom profiling/monitoring tool
  - write a compiler - you need to emit IL code

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)
    (as we will see in the demo)
- you MAY need it to:
  - understand some machine generated code (P/Invoke stubs) or dumps
  - write specialized code using IL instructions not emitted by C#/F# compiler:
    - like **`tail`** - but... JIT does it even without it
    - like **`calli`** and **`ldftn`** - but... in C# 9 there are exposed as *function pointers*
    - use **`Reflection.Emit`** for the above - Mobius example
  - write code omitting default type safety
    - like **`Unsafe`** class
  - write code weaving/IL rewritting - like Fody or any custom profiling/monitoring tool
  - write a compiler - you need to emit IL code
  - just understand how the abstract .NET virtual machine works

Is is worth to know CIL?

- you DON'T need it to:
  - write low-level, *magically* fast code - it is not x64 assembly, JIT will do the work (mostly)
    (as we will see in the demo)
- you MAY need it to:
  - understand some machine generated code (P/Invoke stubs) or dumps
  - write specialized code using IL instructions not emitted by C#/F# compiler:
    - like `tail` - but... JIT does it even without it
    - like `calli` and `ldftn` - but... in C# 9 there are exposed as *function pointers*
    - use `Reflection.Emit` for the above - Mobius example
  - write code omitting default type safety
    - like `Unsafe` class
  - write code weaving/IL rewritting - like Fody or any custom profiling/monitoring tool
  - write a compiler - you need to emit IL code
  - just understand how the abstract .NET virtual machine works
  - for... fun 😇

.NET runtimes

- .NET Framework
- .NET Core 1.0-3.1/.NET 5-6
- Mono (runs Blazor/Xamarin/Unity)
- RIP .NET Compact Framework & .NET Micro Framework
- Rotor/SSCLI Shared Source Common Language Infrastructure
- DotNetAnywhere by Chris Bacon (initial prototype of Blazor)
- DotGNU & Portable .NET (...)
- CoreRT

.NET runtime & C++

C++ piece of the .NET runtime is not-so-obvious:

- .NET Core - more and more parts are being ported to C# (better maintainability, GC pauses/less managed-unmanaged transitions)
  - port JIT and GC to C# issue
  - `stdelemref` and `ldelemaref` JIT helpers to C# (dotnet/runtime#32722)
  - moving portions of the `unbox` helper to C# (dotnet/runtime#32353)
  - port `ThreadPool`
  - port sorting primitives sort
  - rewrite `Enum.CompareTo` in C# (dotnet/runtime#27792)
  - ...

.NET runtime & C++

C++ piece of the .NET runtime is not-so-obvious:

- .NET Core - more and more parts are being ported to C# (better maintainability, GC pauses/less managed-unmanaged transitions)
  - port JIT and GC to C# issue
  - **`stdelemref`** and **`ldelemaref`** JIT helpers to C# (dotnet/runtime#32722)
  - moving portions of the **`unbox`** helper to C# (dotnet/runtime#32353)
  - port **`ThreadPool`**
  - port sorting primitives sort
  - rewrite **`Enum.CompareTo`** in C# (dotnet/runtime#27792)
  - ...
- CoreRT... - many parts rewritten to C# (like type system)
  - still, crucial parts written in C++ - GC, JIT, ...

Materials

- [Taking a look at the ECMA-335 Standard for .NET]
- [Deep-dive into .NET Core primitives: deps.json, runtimeconfig.json, and dll's]
- [Tiered Compilation]
- *"Expert .NET 2.0 IL Assembler"* by Serge Lidin
- *"Compiling for the .Net Common Language Runtime (CLR)"* by John Gough

DEMO