




# Overview of the .NET





# Overview

- Compilation and interpretation
- Virtual machines
- Simple C# program
- CIL, ildasm util
- CLR
- .NET Framework
- JIT, NGEN
- CLS
- .NET 6
- Compare C ++, C # (.NET) method call performance



# Compilation (Ahead-of-Time) and interpretation

- A program written in a high level language can run in two ways
  - Compiled into a program in the native machine language and then run on the target machine.
  - Directly interpreted and the execution is simulated within an interpreter.

# Compilation and interpretation

- How is a C++ program executed on linprog?
  - *cl try.cpp* □ compiling the program into machine code
  - *Try.exe* □ running the machine code
- How is a JavaScript program executed?
  - *cscript.exe try.js*
  - The program just runs, no compilation phase
  - The program *cscript* is the software environment that understands JavaScript language. The program *try.js* is executed (interpreted) within the environment.
- In general, which approach is more efficient?

# Compilation and interpretation

- In general, which approach is more efficient?
  - $A[i][j] = 1;$

## Compilation:

```
mov eax, DWORD PTR _i$[ebp]
imul eax, 20
lea ecx, DWORD PTR _A$[ebp+eax]
mov edx, DWORD PTR _j$[ebp]
mov DWORD PTR [ecx+edx*4], 1
```

## Interpretation:

- create a software environment that understand the language
- put 1 in the array entry  $A[i][j]$ ;

# Compilation and interpretation

- In general, which approach is more efficient?

- $A[i][j] = 1;$

## Compilation:

```
mov eax, DWORD PTR _i$[ebp]
imul eax, 20
lea ecx, DWORD PTR _A$[ebp+eax]
mov edx, DWORD PTR _j$[ebp]
mov DWORD PTR [ecx+edx*4], 1
```

## Interpretation:

- create a software environment that understand the language
- put 1 in the array entry  $A[i][j]$ ;
- For the machine to put 1 in the array entry  $A[i][j]$ , that code sequence still needs to be executed.
- Most interpreter does a little more than the barebone “real work.”

- **Compilation is always more efficient!!**
- **Interpretation provides more functionality.** E.g. for debugging  
One can modify the value of a variable during execution.



# Compilers versus Interpreters

- Compilers “try to be as smart as possible” to fix decisions that can be taken at compile time to avoid to generate code that makes this decision at run time
  - Type checking at compile time vs. runtime
  - Static allocation
  - Static linking
  - Code optimization
- Compilation leads to better performance in general
  - Allocation of variables without variable lookup at run time
  - Aggressive code optimization to exploit hardware features

# Compilers versus Interpreters

## ■ Benefit of interpretation?

- Interpretation facilitates interactive debugging and testing
  - Interpretation leads to better diagnostics of a programming problem
  - Procedures can be invoked from command line by a user
  - Variable values can be inspected and modified by a user
- Some programming languages cannot be purely compiled into machine code alone
  - Some languages allow programs to rewrite/add code to the code base dynamically
  - Some languages allow programs to translate data to code for execution (interpretation)
  - JavaScript Eval() function

```
var x = 10;
var y = 20;
var a = eval("x * y") + " ";
var b = eval("2 + 2") + " ";
var c = eval("x + 17") + " ";
var res = a + b + c;
The result of res will be: "200 4 27 "
```



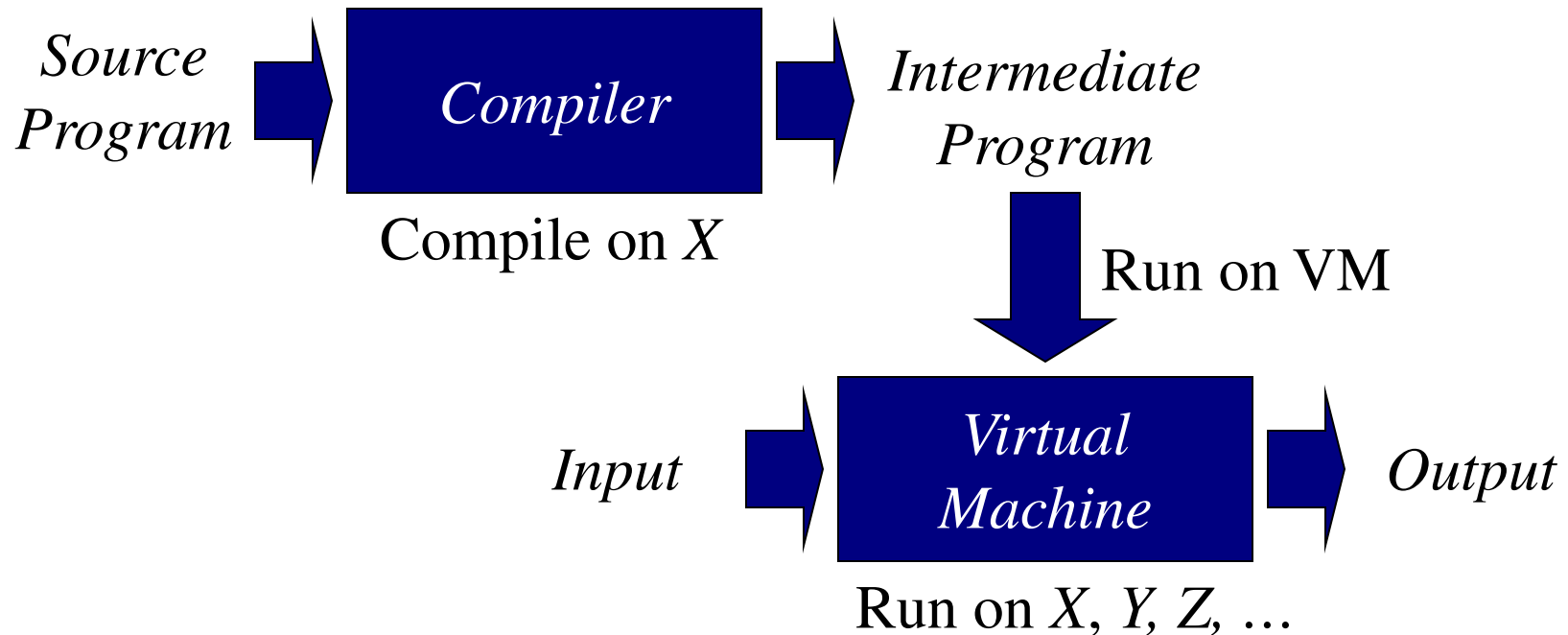


# Virtual Machines (for programming language)

- A virtual machine executes an instruction stream in software
- Adopted by Pascal, Java, Smalltalk-80, C#, functional and logic languages, and some scripting languages
  - Pascal compilers generate P-code that can be interpreted or compiled into object code
  - Java compilers generate bytecode that is interpreted by the Java virtual machine (JVM)
  - C#, VB.NET compilers generate CIL (**Common Intermediate Language**) that is interpreted by the CLR virtual machine
  - The CLR may translate CIL into machine code by just-in-time (JIT) compilation

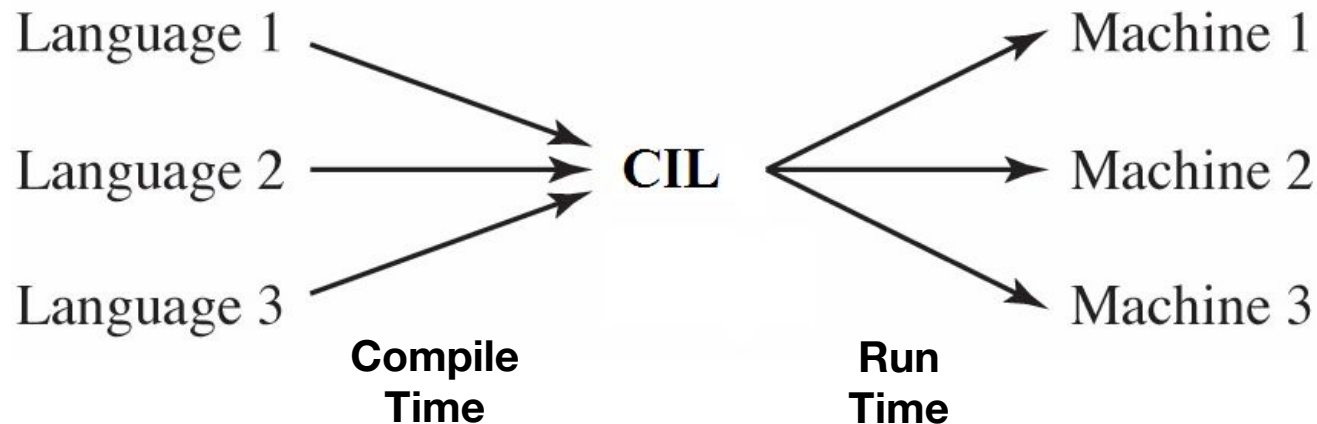
# Compilation and Execution on Virtual Machines

- Compiler generates intermediate program
- Virtual machine interprets the intermediate program



# Two Steps Compilation Process

- Compilation is done in two steps:
  - At compile time: compile each language (C#, VB.Net, C++, etc) to Common Intermediate Language (CIL)
  - At runtime: Common Language Runtime (CLR) uses a Just In Time (JIT) compiler to compile the CIL code to the native code for the device used



# Simple C# program

```
namespace SimpleConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            int init =10;
            int rate =5;
            int pos = init + rate * 60;
            System.Console.WriteLine(pos);
        }
    }
}
```

# C# -> CIL Using ildasm

```
.method private hidebysig static void Main(string[] args) cil managed
{ .entrypoint
  .maxstack 3
  .locals init ([0] int32 'init', [1] int32 rate, [2] int32 pos)
  ldc.i4.s 10
  stloc.0
  ldc.i4.5
  stloc.1
  ldloc.0
  ldloc.1
  ldc.i4.s 60
  mul
  add
  stloc.2
  ldloc.2
  call void [mscorlib]System.Console::WriteLine(int32)
  ret }
```

.maxstack 3

.locals init ([0] int32 'init', [1] int32 rate, [2] int32 pos)

| Local Variables |     |   |
|-----------------|-----|---|
| 0 (init)        | int | 0 |
| 1 (rate)        | int | 0 |
| 2 (pos)         | int | 0 |

| Stack  |  |
|--------|--|
| unused |  |
| unused |  |
| unused |  |

ldc.i4.s 10

| Local Variables |     |   |
|-----------------|-----|---|
| 0 (init)        | int | 0 |
| 1 (rate)        | int | 0 |
| 2 (pos)         | int | 0 |

| Stack  |     |
|--------|-----|
| unused |     |
| unused |     |
| 10     | int |

| Local Variables |     |   |
|-----------------|-----|---|
| 0 (init)        | int | 0 |
| 1 (rate)        | int | 0 |
| 2 (pos)         | int | 0 |

| Stack  |     |
|--------|-----|
| unused |     |
| unused |     |
| 10     | int |

stloc.0

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 0  |
| 2 (pos)         | int | 0  |

| Stack  |  |
|--------|--|
| unused |  |
| unused |  |
| unused |  |

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 0  |
| 2 (pos)         | int | 0  |

| Stack  |  |
|--------|--|
| unused |  |
| unused |  |
| unused |  |

Idc.i4.5

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 0  |
| 2 (pos)         | int | 0  |

| Stack  |     |
|--------|-----|
| unused |     |
| unused |     |
| 5      | int |



| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 0  |
| 2 (pos)         | int | 0  |

| Stack  |     |
|--------|-----|
| unused |     |
| unused |     |
| 5      | int |

stloc.1

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack  |  |
|--------|--|
| unused |  |
| unused |  |
| unused |  |

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack  |  |
|--------|--|
| unused |  |
| unused |  |
| unused |  |

ldloc.0

ldloc.1

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack  |     |
|--------|-----|
| unused |     |
| 5      | Int |
| 10     | int |

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack  |     |
|--------|-----|
| unused |     |
| 5      | int |
| 10     | int |

ldc.i4.s 60

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack |     |
|-------|-----|
| 60    | int |
| 5     | int |
| 10    | int |

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack |     |
|-------|-----|
| 60    | int |
| 5     | int |
| 10    | int |

mul

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack  |     |
|--------|-----|
| unused |     |
| 300    | int |
| 10     | int |

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack  |     |
|--------|-----|
| unused |     |
| 300    | int |
| 10     | int |

add

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack  |     |
|--------|-----|
| unused |     |
| unused |     |
| 310    | int |

| Local Variables |     |    |
|-----------------|-----|----|
| 0 (init)        | int | 10 |
| 1 (rate)        | int | 5  |
| 2 (pos)         | int | 0  |

| Stack  |     |
|--------|-----|
| unused |     |
| unused |     |
| 310    | int |

stloc.2

| Local Variables |     |     |
|-----------------|-----|-----|
| 0 (init)        | int | 10  |
| 1 (rate)        | int | 5   |
| 2 (pos)         | int | 310 |

| Stack  |  |
|--------|--|
| unused |  |
| unused |  |
| unused |  |

| Local Variables |     |     |
|-----------------|-----|-----|
| 0 (init)        | int | 10  |
| 1 (rate)        | int | 5   |
| 2 (pos)         | int | 310 |

| Stack  |  |
|--------|--|
| unused |  |
| unused |  |
| unused |  |

## ldloc.2

| Local Variables |     |     |
|-----------------|-----|-----|
| 0 (init)        | int | 10  |
| 1 (rate)        | int | 5   |
| 2 (pos)         | int | 310 |

| Stack  |     |
|--------|-----|
| unused |     |
| unused |     |
| 310    | int |

| Local Variables |     |     |
|-----------------|-----|-----|
| 0 (init)        | int | 10  |
| 1 (rate)        | int | 5   |
| 2 (pos)         | int | 310 |

| Stack  |     |
|--------|-----|
| unused |     |
| unused |     |
| 310    | int |

call void mscorlib[System.Console::WriteLine(int32)  
ret

| Local Variables |     |     |
|-----------------|-----|-----|
| 0 (init)        | int | 10  |
| 1 (rate)        | int | 5   |
| 2 (pos)         | int | 310 |

| Stack  |  |
|--------|--|
| unused |  |
| unused |  |
| unused |  |





# **Common Intermediate Language (CIL)**

- Much like the native languages of devices.
- CIL was originally known as Microsoft Intermediate Language (MSIL).
- CIL is a CPU- and platform-independent instruction set.
- It can be executed in any environment supporting the .NET framework

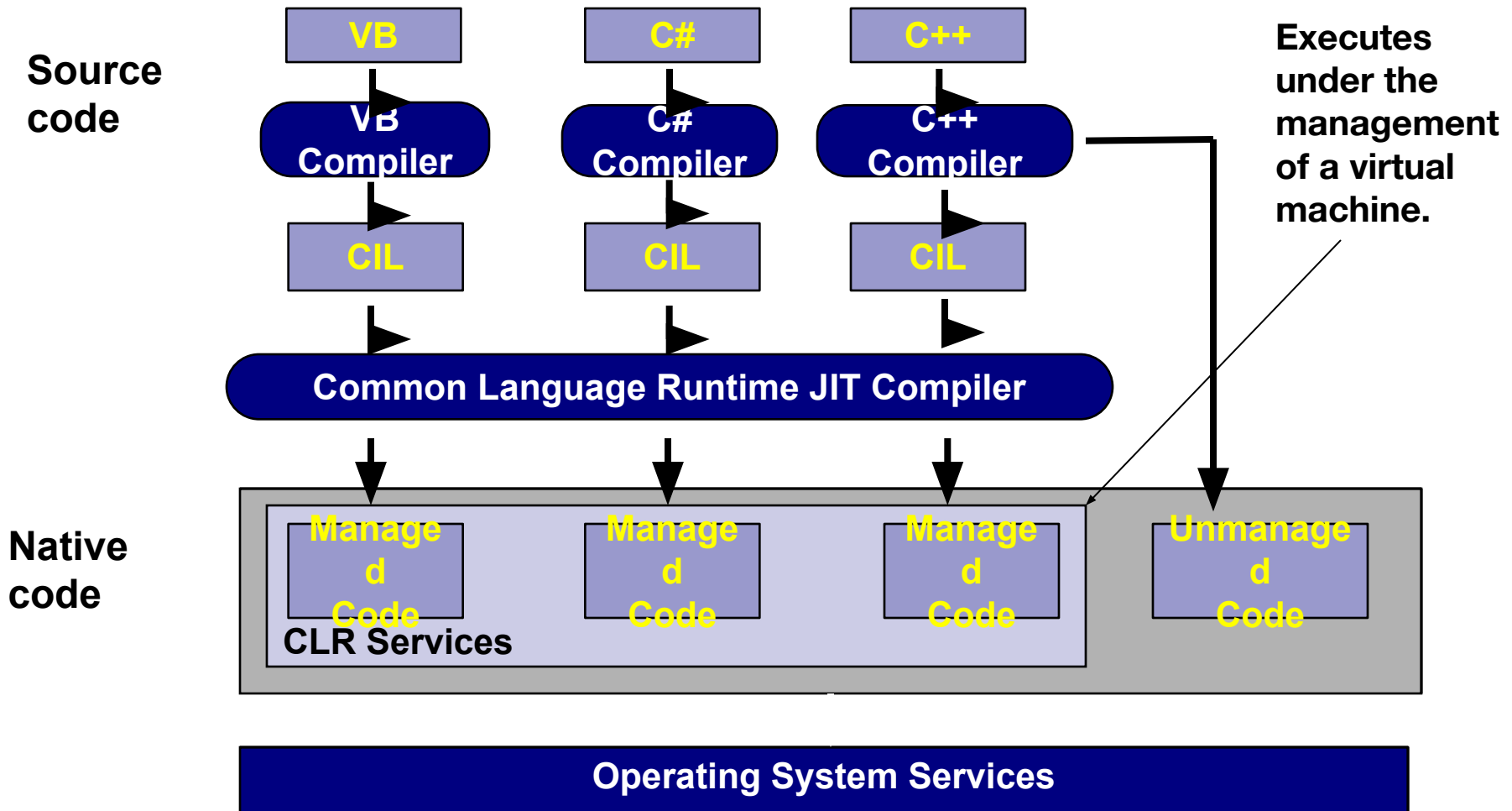


# Common Language Runtime (CLR)

- The Common Language Runtime (CLR) manages the execution of code.
- CLR uses Just-In-Time (JIT) compiler to compile the CIL code to the native code for device used.
- Through the runtime compilation process CIL code is verified for safety during runtime, providing better security and reliability than natively compiled binaries.
- Native image generator compilation (NGEN) can be used to produces a native binary image for the a specific environment. What is the point?

# Compilation Process

So if we have 3 programming languages and 3 devices, how many compilers do we need?





# Platform and Language Independent

- What we have described so far will lead us to Platform independent environment. How?
- Can we use compiled classes written in X language in a program written in Y language?
- VB.NET + C#.NET code

# Language interoperability

- All .NET languages can interoperate

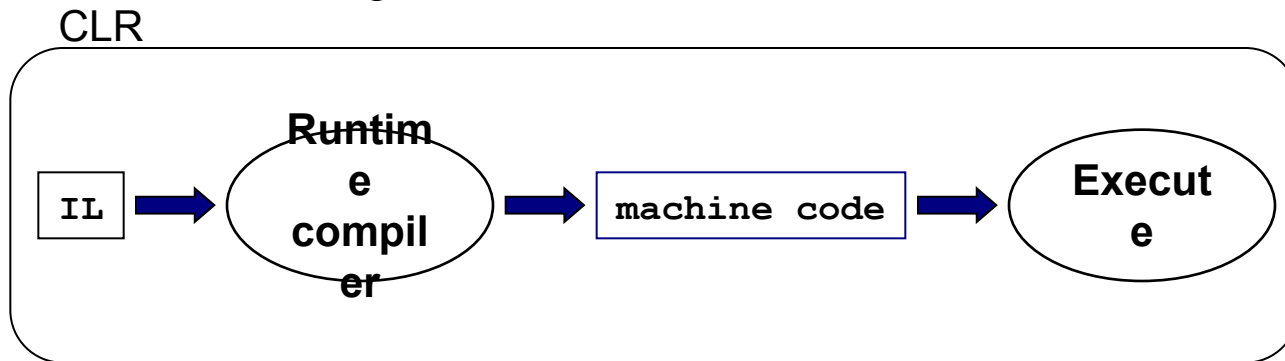
C# calling  
VB.NET

```
class Hello
{
    static void Main()
    {
        System.Console.WriteLine(Greeting.Message());
    }
}
```

```
Class Greeting
    Shared Function Message() As String
        Return "hello"
    End Function
End Class
```

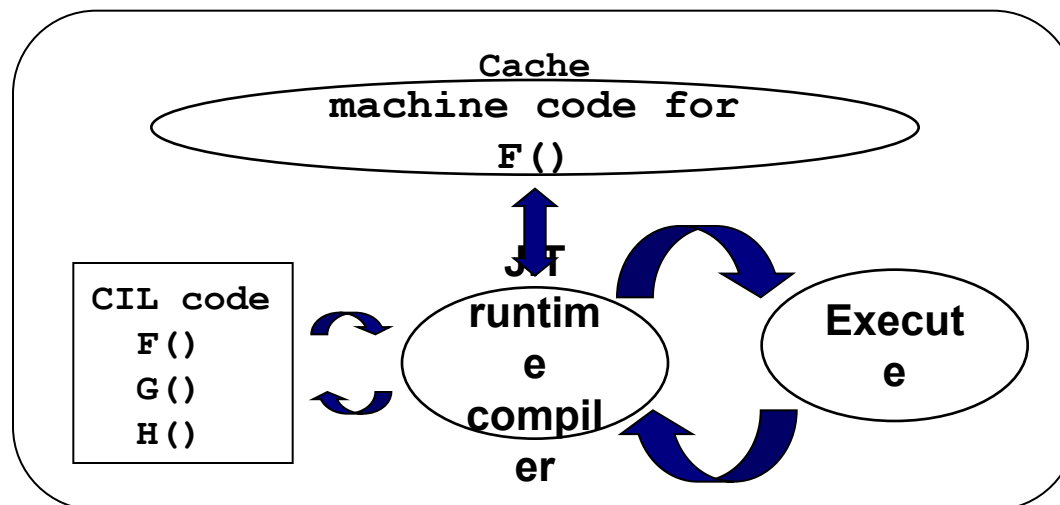
# Execution engine

- *Common Language Runtime (CLR)* is the execution engine
  - loads IL
  - compiles IL
  - executes resulting machine code



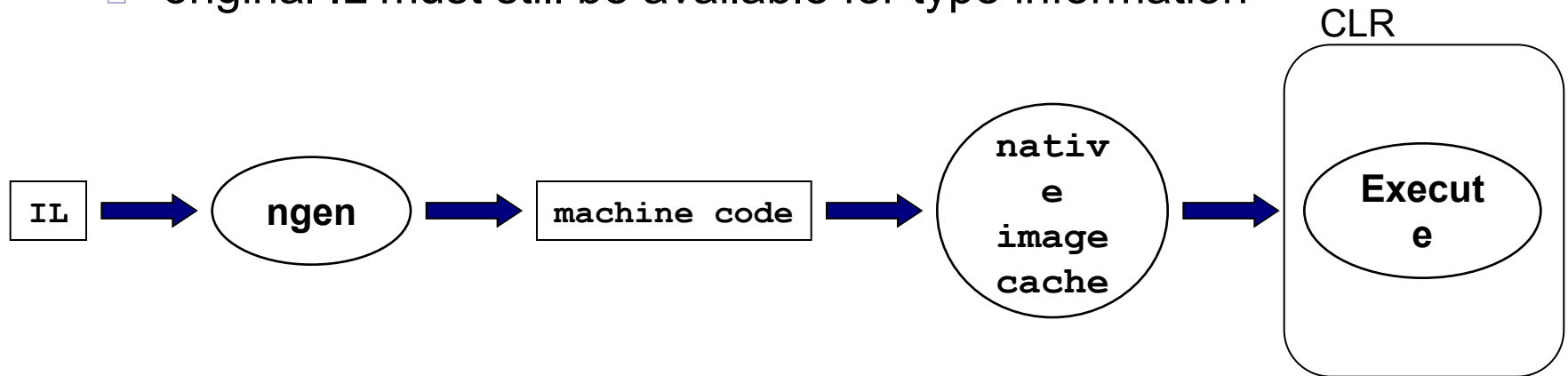
# JIT runtime compile

- CIL is compiled into machine code at runtime by the CLR
  - compiles methods as needed
  - called *just in time* (JIT) compile
- JIT compilation model:
  - first time method is called the IL is compiled and optimized
  - compiled machine code is cached in transient memory
  - cached copy used for subsequent calls



# NGEN install time compile

- Can compile CIL into machine code when app installed
  - use native image generator **ngen.exe**
  - can speed startup time since code pre-compiled
  - but cannot do as many optimizations
  - original IL must still be available for type information





# Language variability

Not all .NET languages have exactly the same capabilities

differ in small but important ways

C#

signed integer →  
unsigned integer →

```
class Hello
{
    static void Main()
    {
        int i;
        uint u;
    }
}
```

VB.NET

signed integer only →

```
Class Greeting
    Shared Sub Main()
        Dim i as Integer
    End Sub
End Class
```

# Common Language Specification

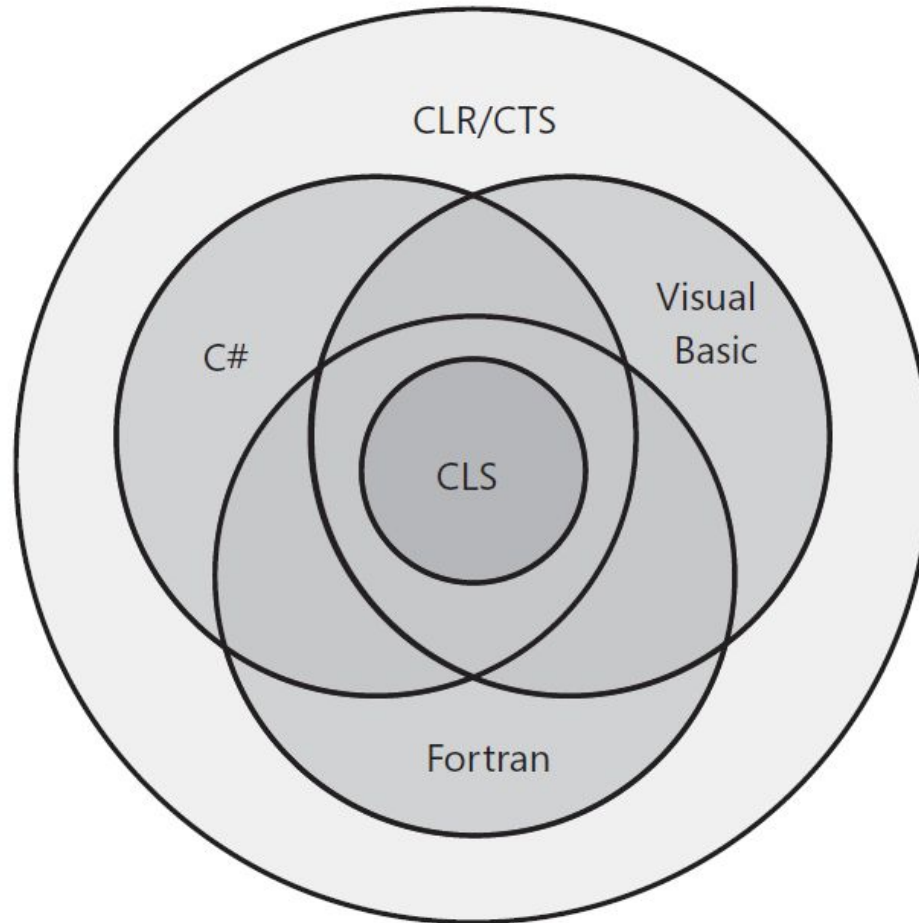
- Common Language Specification (CLS) defines type subset
  - required to be supported by all .NET languages
  - limiting code to CLS maximizes language interoperability
  - code limited to CLS called *CLS compliant*

not CLS compliant  
to use `uint` in public  
interface of public class



```
public class Calculator
{
    public uint Add(uint a, uint b)
    {
        return a + b;
    }
}
```

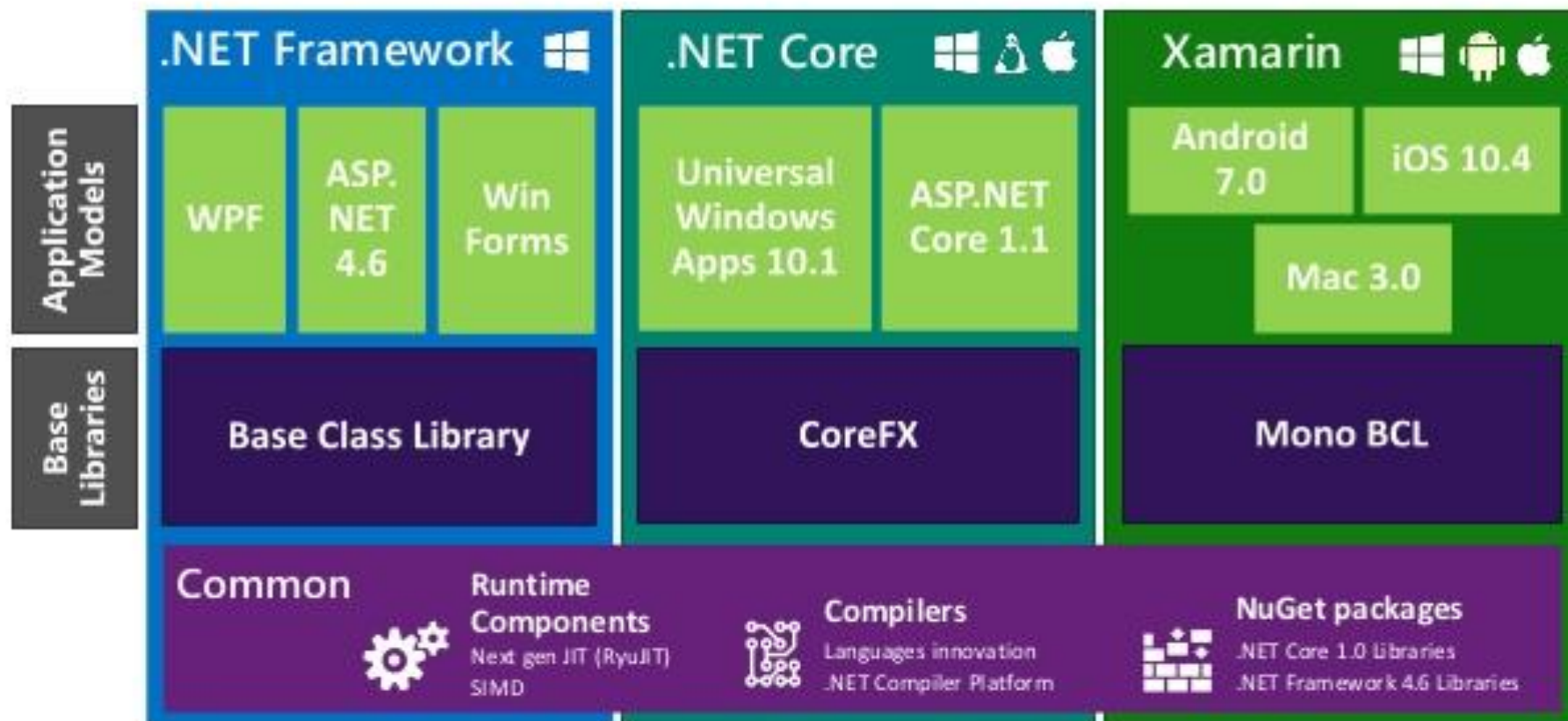
# CLS,CLR/CTS & Languages



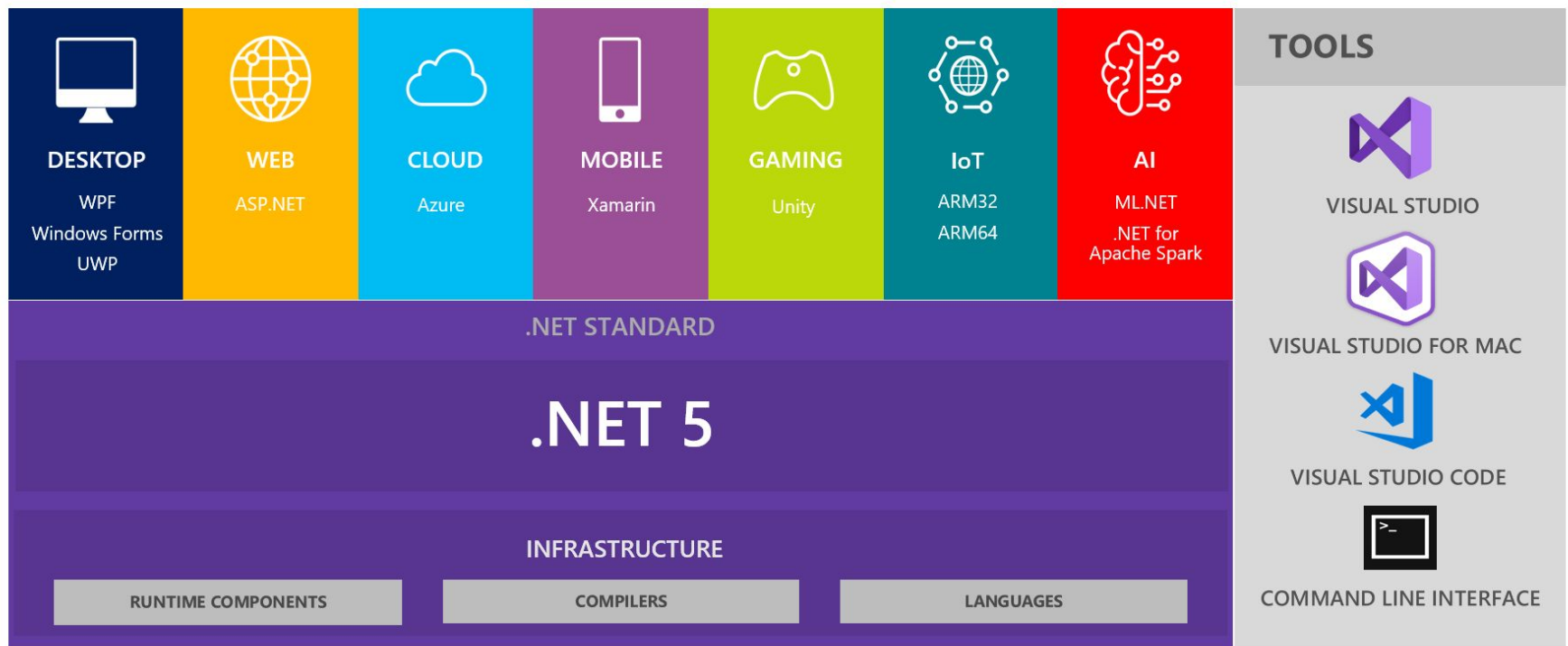
Languages offer a subset of the CLR/CTS and a superset of the CLS (but not necessarily the same superset).

# The big picture of .NET Platforms

DOT  
NEXT



# .NET – A unified platform



# .NET Schedule



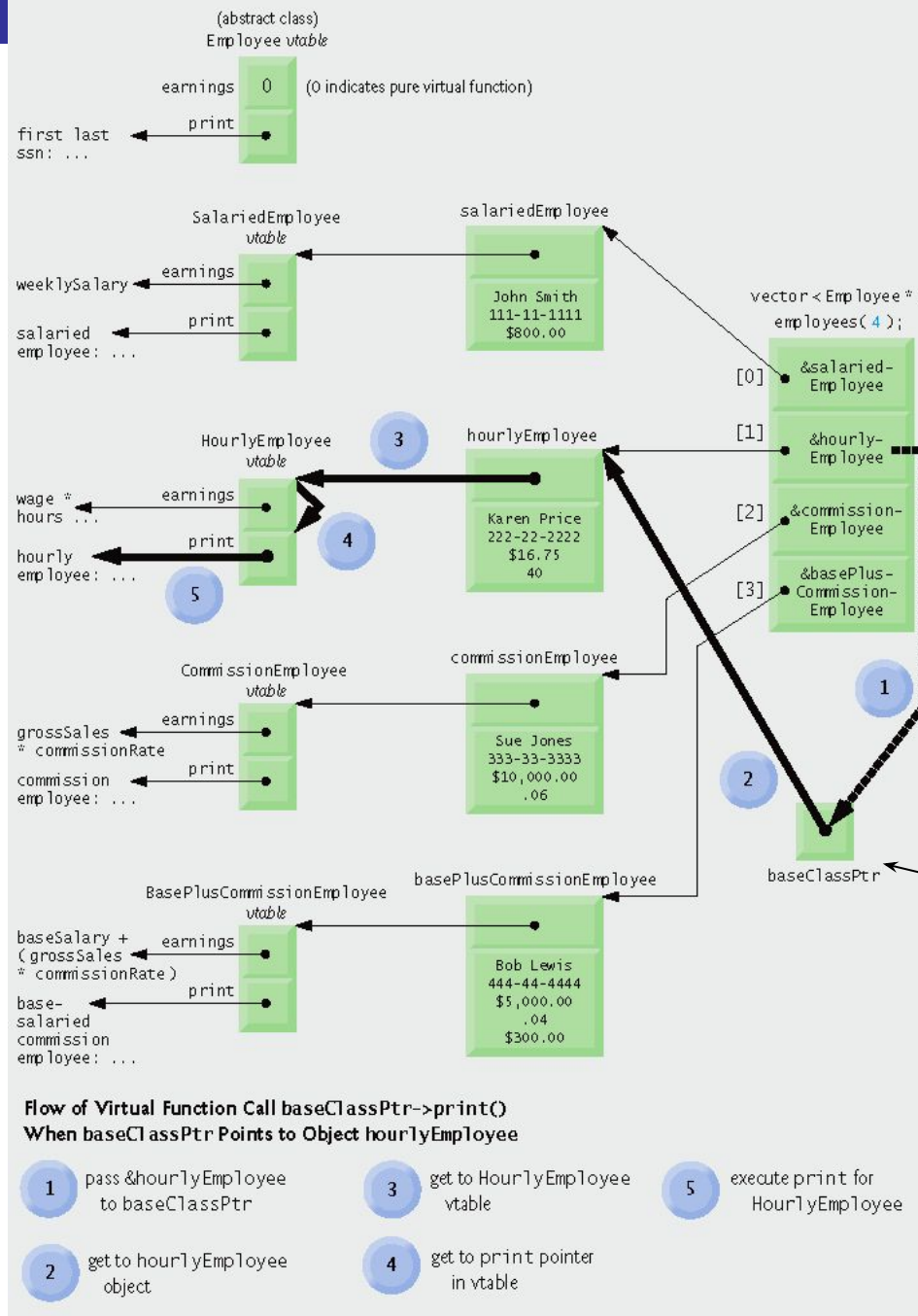
- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed



# Method call performance

Let's compare C ++, C # (.NET) method call performance

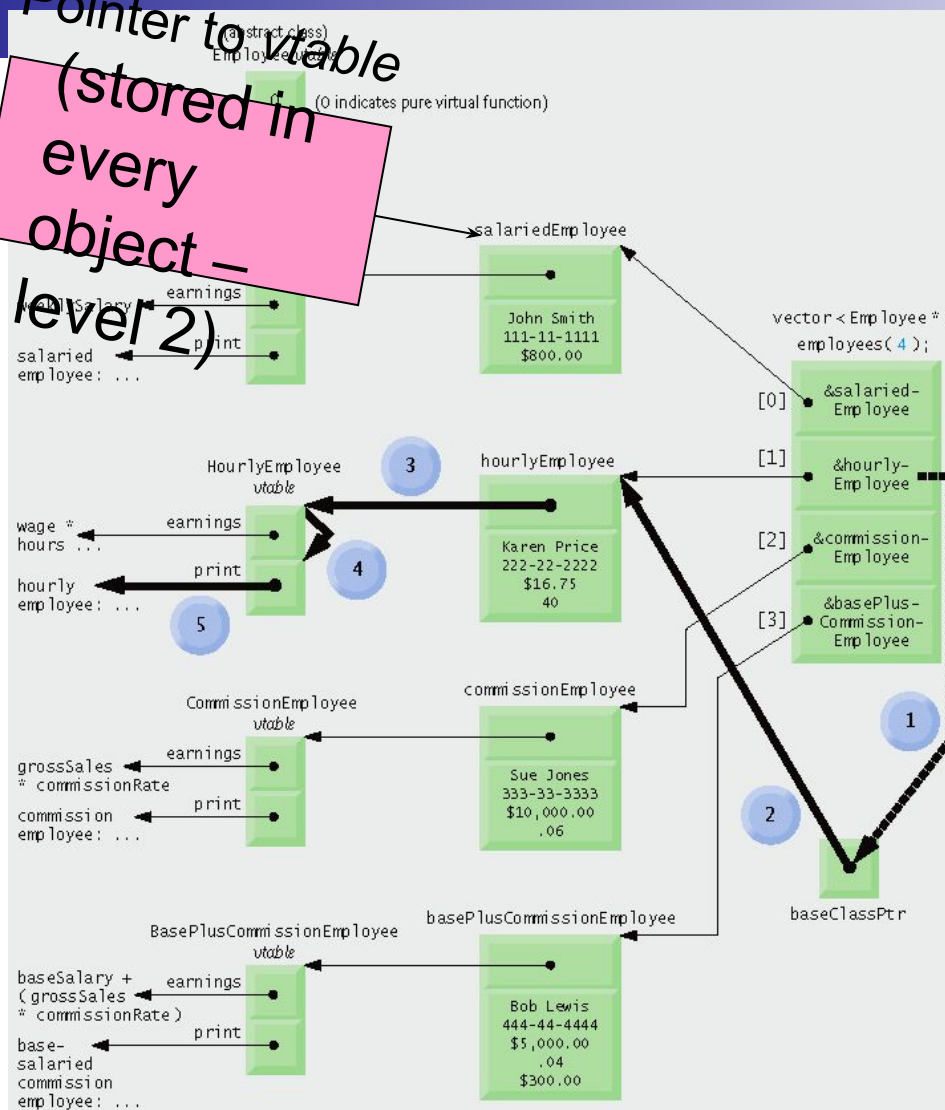
- C++ Function
- C++ Virtual Function
- C# (.NET) Method



Pointer to  
object  
(level 1)



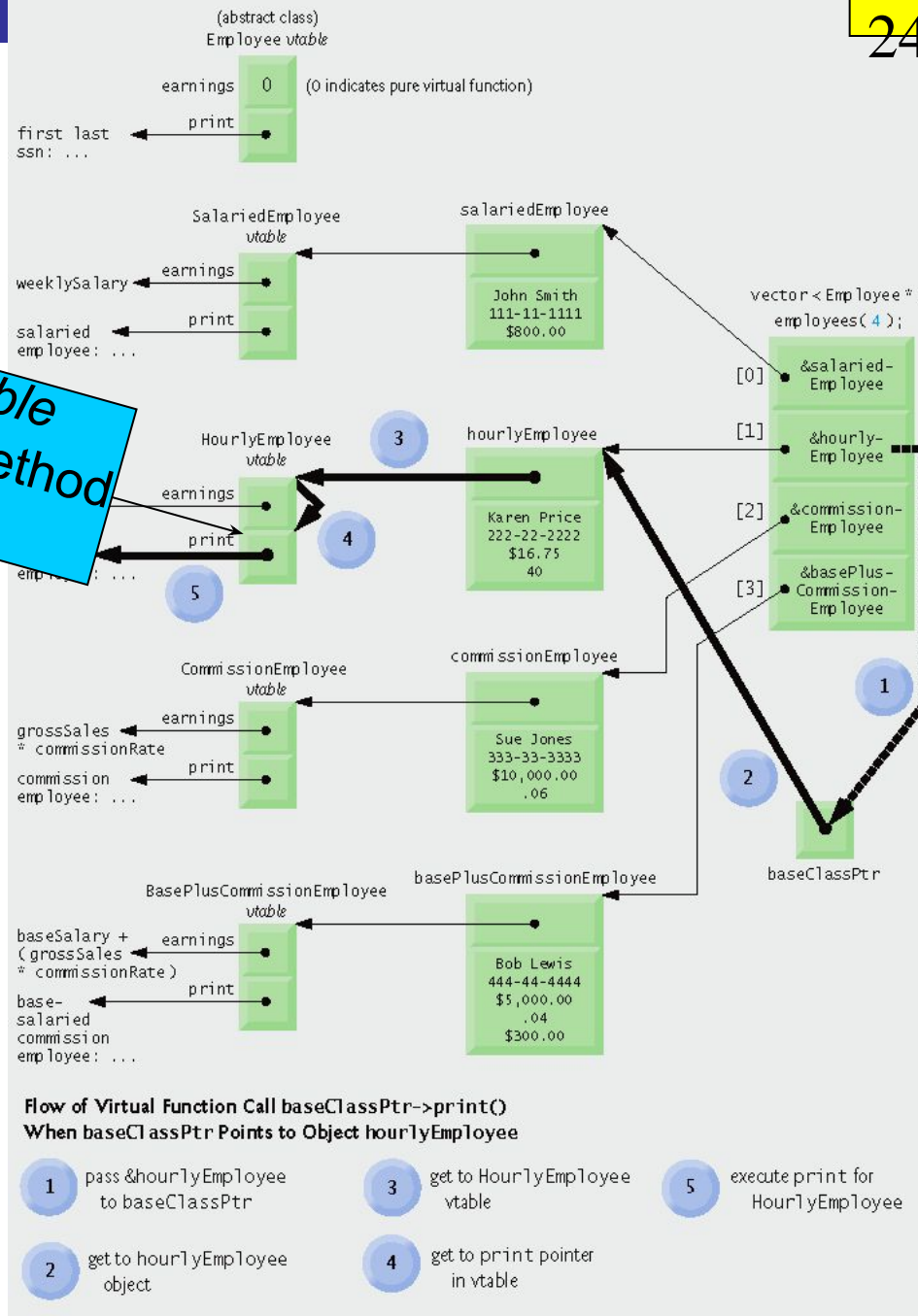
**Pointer to vtable  
(stored in  
every  
object -  
level 2)**



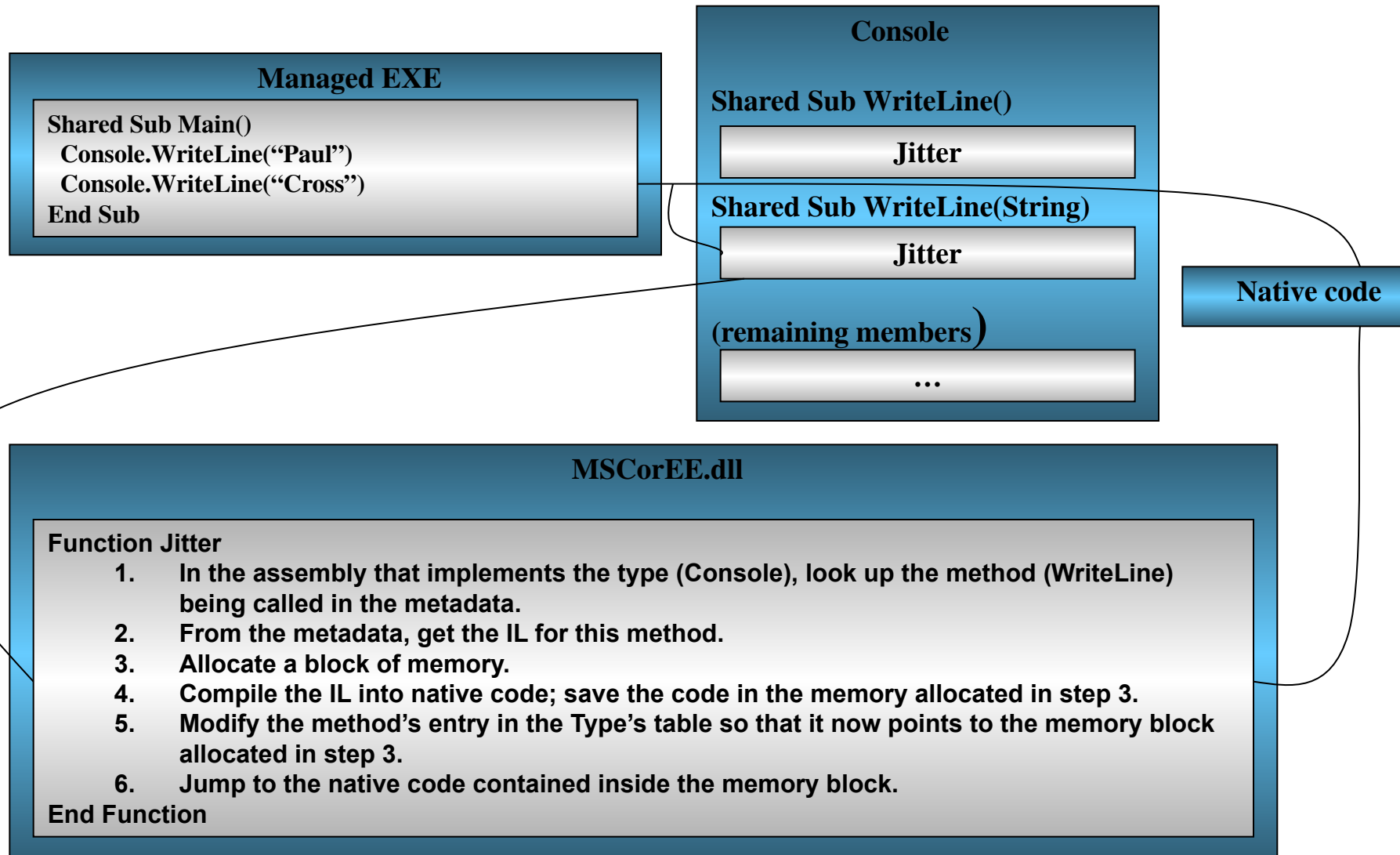
**Flow of Virtual Function Call baseClassPtr->print()  
When baseClassPtr Points to Object hourlyEmployee**

- 1 pass `&hourlyEmployee` to `baseClassPtr`
- 2 get to `hourlyEmployee` object
- 3 get to `HourlyEmployee` vtable
- 4 get to `print` pointer in vtable
- 5 execute `print` for `HourlyEmployee`

Index into vtable  
to access method  
(level 3)



# Calling a method for the first time





# Performance Impact

| Call Number | C++ | C++ Virtual    | .NET                          |
|-------------|-----|----------------|-------------------------------|
| 1           | X   | X + 2 pointers | X+ 2 pointers+<br>JIT compile |
| 2,3.....    | X   | X + 2 pointers | X+ 2 pointers                 |