The CLR and IL in a Nutshell

Bart J.F. De Smet
bartde@outlook.com

pluralsight
hardcore dev and IT training

A Whirlwind Tour…

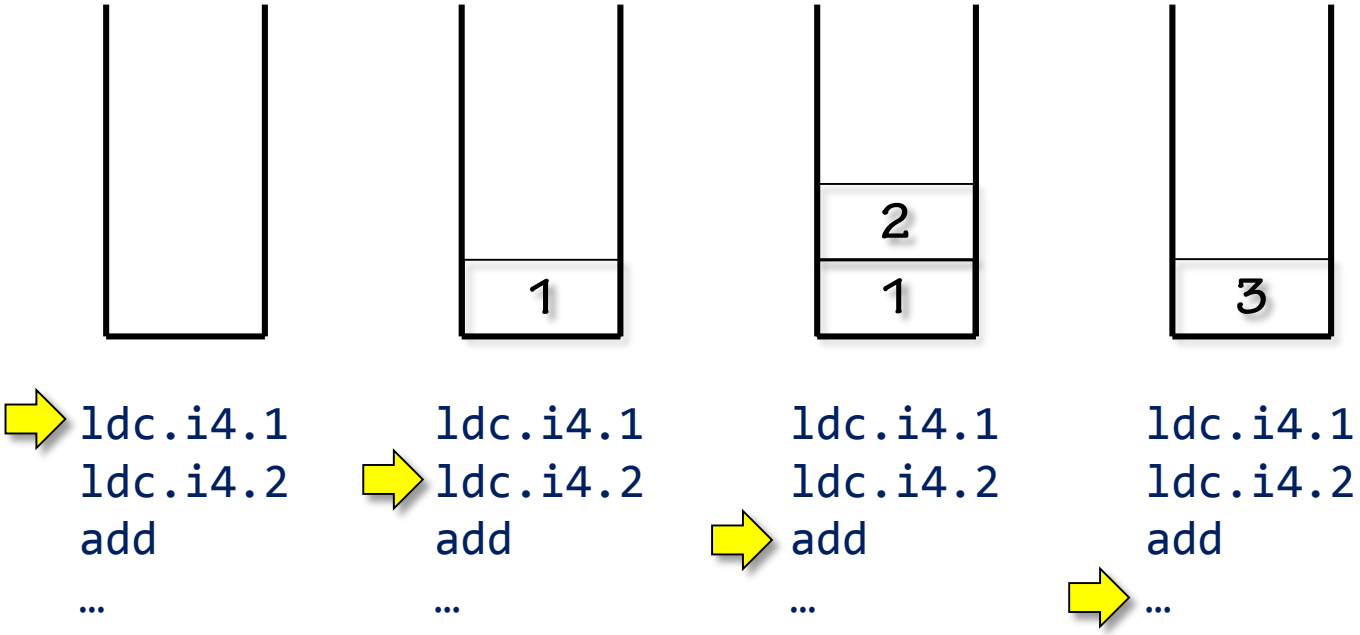
- **Quick refresher of:**
  - CLR internals
  - IL instructions
- **Check out our course catalog!**
  - CLR Fundamentals, by Mike Woodring
  - MSIL for the C# Developer, by Filip Ekberg
- **Books**
  - CLR via C#, Jeffrey Richter
  - Expert .NET 2.0 IL Assembler, Serge Lidin
  - Shared Source CLI 2.0 Internals, Joel Poebar & Ted Neward

Quick Intro to IL Code

- **Intermediate Language (IL)**
  - Virtual Machine language of the CLR
    - Emitted by managed language compilers (C#, VB, F#, etc.)
    - NGEN or JIT compiled to native code
  - Stack-based evaluation
    - No registers
    - Locals, arguments, fields, etc.
  - Verification
    - Type safe
    - Memory safe
  - Leverages metadata

Quick Intro to IL Code

- **Stack-based evaluation**
  - "Scratch-pad" for computation
  - Pop *operands*, execute *operator*, push result
  - Each instruction has a net effect on the stack
    - E.g. add = 2 x pop +1 x push
  - Stack transitions for "1 + 2"



Quick Intro to IL Code

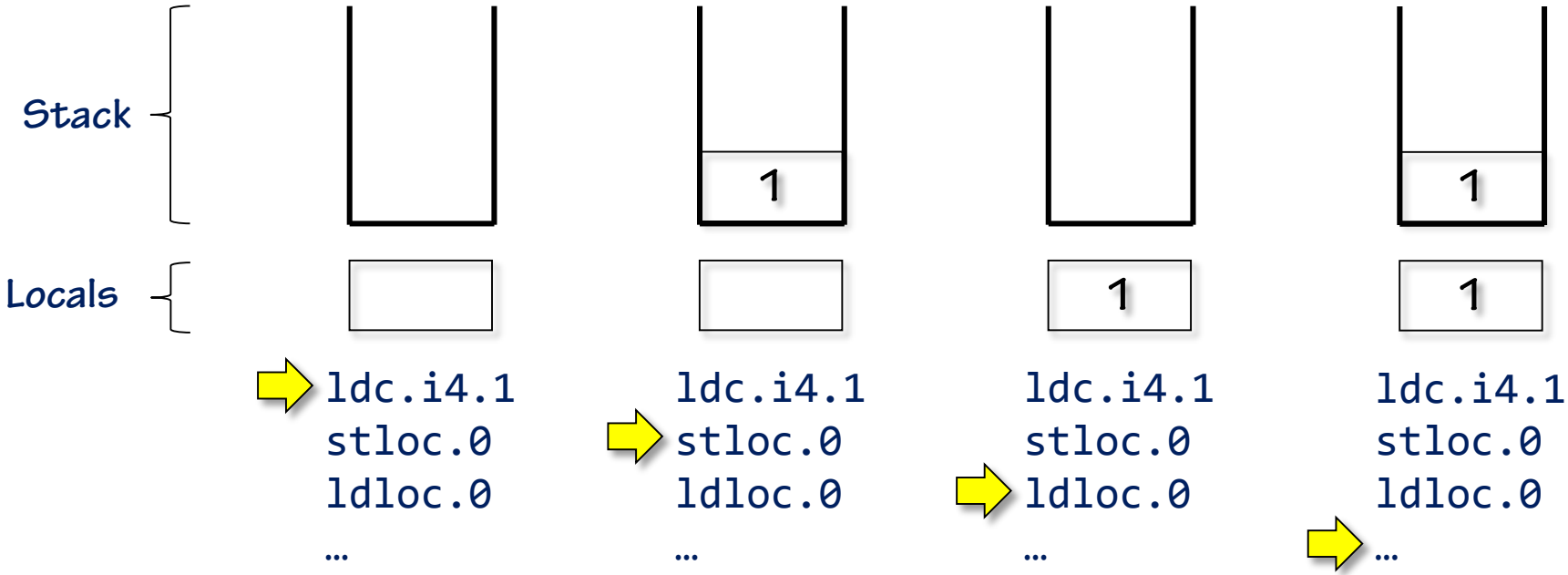
- **Essential instructions**
  - pop
    - Pops the object on top of the stack
    - Often used to discard stuff to rebalance the stack
  - dup
    - Duplicates the object on top of the stack
    - Often used to eliminate loads and stores to locals
  - nop
    - No-operation, doesn't do anything "useful"
    - Often used in non-optimized builds (csc /o-)
      - Breakpoints can only be set on instructions
      - E.g. emit nop for lines with curly braces

Quick Intro to IL Code

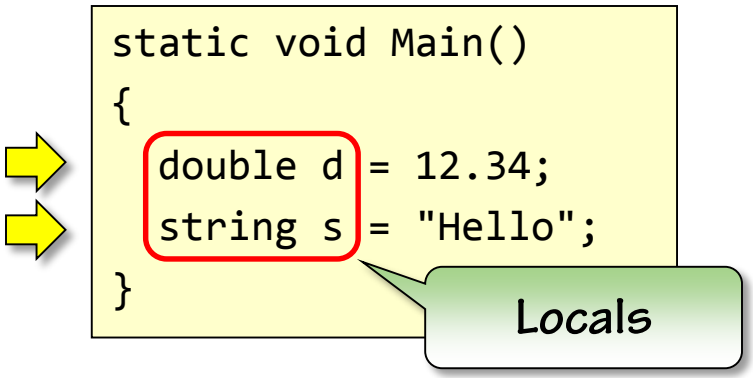
- **Loading constants**
  - Numerical:
    - ldc.i4 for “load constant integer 4 bytes” (int)
    - ldc.r8 for “load constant floating point 8 bytes” (double)
    - Value as an operand in the IL instruction stream
    - Shorthand instructions, e.g. ldc.i4.1 for Int32 1, ldc.i4.m1 for Int32 -1
  - Boolean:
    - Represented as 0 (false) or 1 (true)
  - String:
    - ldstr for “load string”
    - Value as an operand that points to a string table entry
  - Null:
    - ldnull
    - Null reference, useful for initialization or cleanup of locations

Quick Intro to IL Code

- **Locals**
  - Typed slots to hold objects
  - JIT may put those in machine registers or on the stack
  - Instructions to load and store
    - E.g. ldloc.0 pushes the 0th local onto the evaluation stack
    - E.g. stloc.1 pops from the evaluation stack and stores into the 1st local



Quick Intro to IL Code




```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  .maxstack  1
  .locals /*11000001*/ init (float64 V_0, string V_1)
  IL_0000:  /* 00   |                  */ nop
  IL_0001:  /* 23   | AE47E17A14AE2840 */ ldc.r8     12.34
  IL_000a:  /* 0A   |                  */ stloc.0
  IL_000b:  /* 72   | (70)000001       */ ldstr      "Hello" /* 70000001 */
  IL_0010:  /* 0B   |                  */ stloc.1
  IL_0011:  /* 2A   |                  */ ret
}
```

Op codes

Quick Intro to IL Code

- **Arithmetic, relational, logical, conversions**
  - Arithmetic instructions
    - neg, add, sub, mul, div, rem, shl, shr
    - Overflow variants with .ovf suffix (cf. checked in C#)
  - Relational instructions
    - clt, cle, cgt, cge, ceq, cne (also branch counterparts)
    - Push zero or one based on outcome of comparison
  - Bitwise logical instructions
    - and, or, xor, not
    - Short-circuiting behavior requires control flow
  - Conversions
    - conv.<type>, e.g. conv.i4 to convert to 4-byte integer
    - isinst for type checks
    - castclass for casts to a specified type

Quick Intro to IL Code

- **Branch instructions**
  - Unconditional branch instructions
    - br – method-local branch using the specified offset
  - Conditional branch instructions
    - brtrue, brfalse – check top of stack for true or false
    - beq, bne, blt, ble, bgt, bge – relational operators with branching
  - Variants
    - .s suffix – short branch if offset fits in 1 byte
    - .un – unsigned variants
  - Switch tables
    - switch – jumps based on integer operand

Quick Intro to IL Code

```
static void Gt(int a, int b) {
    if (a > b)
        Console.WriteLine("a > b");
    else
        Console.WriteLine("a <= b");
}
```

Branch

Fall-through

```
.method private hidebysig static void  Gt(int32 a, int32 b) cil managed
{
  IL_0000:  ldarg.0
  IL_0001:  ldarg.1
  IL_0002:  ble.s      IL_000f   /* 31 (ble.s) | 0B (offset) */
  IL_0004:  ldstr      "a > b"
  IL_0009:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000e:  ret
  IL_000f:  ldstr      "a <= b"
  IL_0014:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_0019:  ret
}
```

Short branch

Quick Intro to IL Code

- **Call stacks**
  - Different call instructions
    - call – regular "direct" call
    - callvirt – call with virtual dispatch
    - calli – indirect call through a pointer (interop)
  - Arguments
    - Push arguments to make a method call
      - Arguments passed left-to-right on the stack
      - Instance methods hold "this" in 0th argument
    - Call stack frames hold locals and arguments
    - Instruction to load an argument: ldarg
  - Return to caller using ret instruction
    - Stack should only contain one object (or none if void)
  - Exceptions unwind the call stack

Quick Intro to IL Code

```
static void Main()
{
  double d = Math.Pow(3.14, 2.81);
  Console.WriteLine(d);
}
```

```
.method private hidebysig static void  Main() cil managed
{
  .entrypoint
  .maxstack  2
  .locals init (float64 V_0)
  IL_0000:  ldc.r8     3.1400000000000001
  IL_0009:  ldc.r8     2.8100000000000001
  IL_0012:  call       float64 [mscorlib]System.Math::Pow(float64, float64)
  IL_0017:  stloc.0
  IL_0018:  ldloc.0
  IL_0019:  call       void [mscorlib]System.Console::WriteLine(float64)
  IL_001e:  ret
}
```

Quick Intro to IL Code

- **Calls in C#**
  - Static methods use call
  - Instance methods use callvirt
    - Even for non-virtuals
    - Performs null check for v-table

```
.class Foo {
  .method void Bar(int32 a) {
    ldarg.1
    call void Console::WriteLine(int32)
    ret
  }
}
```

Does not use ldarg.0

- **Special calls**
  - Use native implementation provided by the CLR
  - [MethodImpl(MethodImplOptions.InternalCall)]
    - Often used for reflection-related stuff
    - E.g. System.Object::GetType
  - [DllImport("QCall")]
    - Quick calls
      - from the mscorlib.dll assembly
      - to native helper methods in mscorwks.dll or clr.dll
    - E.g. System.GC::_Collect

Quick Intro to IL Code

- **Throwing exceptions**
  - throw
    - Used for “throw ex;” in C#
    - Resets the stack trace
      - ExceptionDispatchInfo in .NET 4.5
  - rethrow
    - Used for “throw;” in C#

```
try {
  // Do stuff
}
catch (SomeException ex) {
  throw; // rethrow
}
```

- **Handling exceptions**
  - Metadata describes protected regions, i.e. “try”…
    - “catch” – type-based exception handling
    - “finally” – cleanup upon successful or exceptional exit
    - “fault” – not available in C#
    - “filter” – conditional exception handling available in Visual Basic
  - Control flow in and out of a protected region
    - leave instruction to exit a protected region
      - causes handlers to run
    - endfinally and endfault to exit handlers

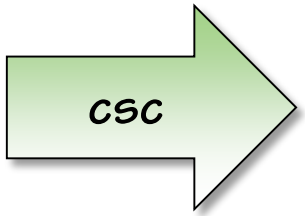
Quick Intro to IL Code

- **Working with objects**
  - newobj – creates a new object on the GC heap
    - Causes memory allocation and can throw OutOfMemoryException
    - Runs specified constructor on object with zeroed memory
    - Returns reference to newly created object
  - ldfld and stfld
    - Loads and stores from/to fields
    - Parameterized by metadata token of the fields
- **Arrays (one-dimensional)**
  - newarr – creates a new array of the specified length
  - ldlen – loads the array length
  - ldelem and stelem – loads and stores array elements using an index

JIT and NGEN

- **Compilation model**
  - Front-end
    - Managed language compilers, such as C#, Visual Basic, F#
    - Emit IL code

```
static int Add(int a, int b)
{
  return a + b;
}
```

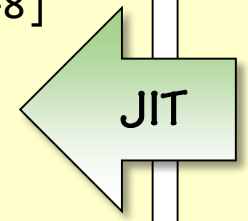



```
.method static int32 Add(
    int32 a, int32 b)
{
  ldarg.0
  ldarg.1
  add
  ret
}
```

JIT and NGEN

- **Compilation model**
  - Front-end
    - Managed language compilers, such as C#, Visual Basic, F#
    - Emit IL code
  - Back-end
    - Just in Time (JIT) compilation to x86, x64, or ARM at runtime, or
    - Native Image Generation (NGEN) ahead of time, e.g. during setup

```
7ff8`1c8100d0  mov  dword ptr [rsp+10h],edx
7ff8`1c8100d4  mov  dword ptr [rsp+8],ecx
7ff8`1c8100d8  mov  ecx,dword ptr [rsp+10h]
7ff8`1c8100dc  mov  eax,dword ptr [rsp+8]
7ff8`1c8100e0  add  eax,ecx
7ff8`1c8100e2  jmp  00007ff8`1c8100e4
7ff8`1c8100e4  nop
7ff8`1c8100e5  ret
```




```
.method static int32 Add(
    int32 a, int32 b)
{
  ldarg.0
  ldarg.1
  add
  ret
}
```

Intermezzo: Compiler Tradeoffs

Frond-end (C#)

- Developer productivity
- Global program knowledge
- Machine agnostic
- More time to optimize
- Can defer to JIT

Back-end (JIT)

- Efficient execution
- Local program knowledge
- Machine knowledge
- Less time to optimize
- Last in line

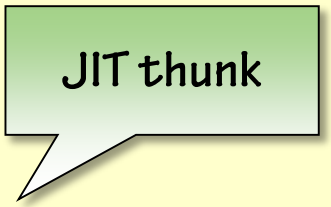
JIT and NGEN

- **Using SOS to analyze a method**
  - Method descriptor (“md”)
    - Internal identifier of the method
    - !name2ee <module> <method>
  - Display method info
    - Class, method table, JIT status, etc.
    - !dumpmd <method desc>
  - Break on method
    - Native instructions only appear after JIT
    - !bpmd command
      - !bpmd -md <method desc>
      - !bpmd <module> <method>
      - !bpmd <source file>:<line> (if PDBs are available)
  - Show code
    - !dumpil <method desc>
    - !U <method desc>

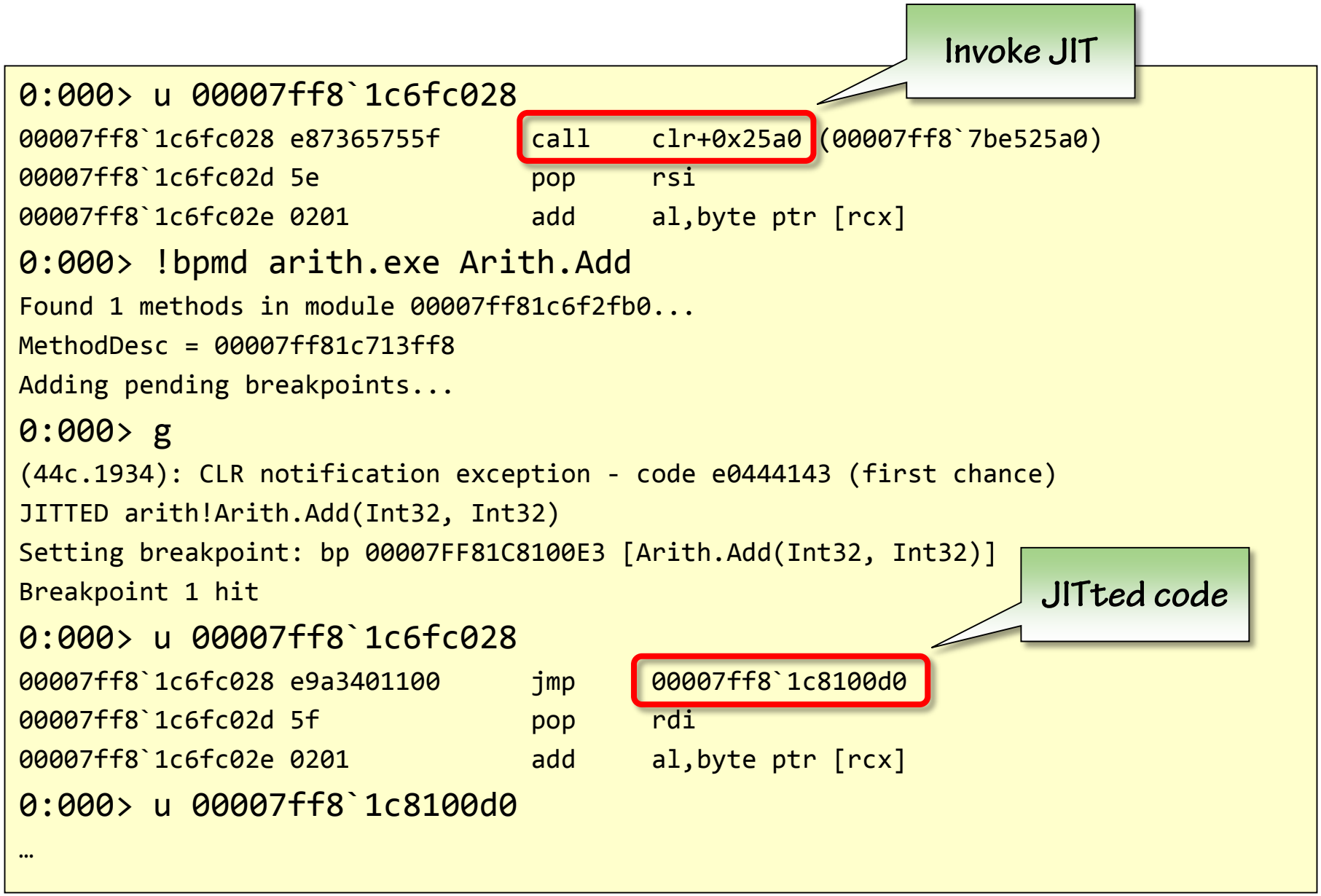
JIT and NGEN

```
C:\Demo> cdb.exe arith.exe
```

```
0:000> !bpmd arith.exe Arith.Main
Found 1 methods in module 00007ff81c6f2fb0...
MethodDesc = 00007ff81c6f3fe8
Adding pending breakpoints...
0:000> g
(44c.1934): CLR notification exception - code e0444143 (first chance)
JITTED arith!Arith.Main()
Setting breakpoint: bp 00007FF81C810090 [Arith.Main()]
Breakpoint 0 hit
0:000> !u 00007ff81c810090
Normal JIT generated code
Arith.Main()
7ff8`1c810090   sub   rsp,28h
7ff8`1c810094   mov   edx,2
7ff8`1c810099   mov   ecx,1
7ff8`1c81009e   call  7ff8`1c6fc028 (Arith.Add(Int32, Int32), mdToken: 000006000002)
7ff8`1c8100a3   mov   ecx,eax
7ff8`1c8100a5   call  mscorlib_ni+0xd24780 (7ff8`7b5d4780) (Console.WriteLine(Int32),
```



JIT and NGEN


Invoke JIT
0:000> u 00007ff8`1c6fc028
00007ff8`1c6fc028 e87365755f call clr+0x25a0 (00007ff8`7be525a0)
00007ff8`1c6fc02d 5e pop rsi
00007ff8`1c6fc02e 0201 add al,byte ptr [rcx]
0:000> !bpmd arith.exe Arith.Add
Found 1 methods in module 00007ff81c6f2fb0...
MethodDesc = 00007ff81c713ff8
Adding pending breakpoints...
0:000> g
(44c.1934): CLR notification exception - code e0444143 (first chance)
JITTED arith!Arith.Add(Int32, Int32)
Setting breakpoint: bp 00007FF81C8100E3 [Arith.Add(Int32, Int32)]
Breakpoint 1 hit
JITted code
0:000> u 00007ff8`1c6fc028
00007ff8`1c6fc028 e9a3401100 jmp 00007ff8`1c8100d0
00007ff8`1c6fc02d 5f pop rdi
00007ff8`1c6fc02e 0201 add al,byte ptr [rcx]
0:000> u 00007ff8`1c8100d0
…

JIT and NGEN

- **Inlining**
  - Quite aggressive by default
  - Using MethodImplAttribute and MethodImplOptions
    - NoInlining – prevents the method from getting inlined
    - NoOptimization – turns off optimizations in JIT / NGEN compilation
    - AggressiveInlining – inlines the method whenever possible

- **JIT intrinisics**
  - Methods provided by the CLR as native code
  - E.g. Math.Sin

- **NGEN**
  - ngen install <assembly>
  - Creates _ni file in native image cache
    - %windir%\assembly\NativeImages_v4.0.30319_32
    - Folder per assembly

JIT and NGEN

```
static void Main() {
  Console.WriteLine(Add(1, 2));
}

static int Add(int a, int b) {
  return a + b;
}
```

```
0:000> !U 00007FFD85E70090
Normal JIT generated code
Arith.Main()
Begin 00007ffd85e70090, size 14
00007ffd`85e70090   sub     rsp,28h
00007ffd`85e70094   mov     ecx,3
00007ffd`85e70099   call    mscorlib_ni+0xd24780 (00007ffd`e4b44780)
   (System.Console.WriteLine(Int32), mdToken: 000000000600099d)
00007ffd`85e7009e   nop
00007ffd`85e7009f   add     rsp,28h
00007ffd`85e700a3   ret
```

1 + 2 = 3

Summary

- **Intermediate Language (IL)**
  - Virtual machine with stack-based evaluation
  - Metadata for types and members
  - Static typing all around
  - Rich instruction set

- **Just-in-time compilation (JIT)**
  - Translation of IL to assembler (x86, x64, ARM)
  - Various optimizations such as inlining
  - Use SOS to inspect generated code
  - Can run ahead of time with NGEN