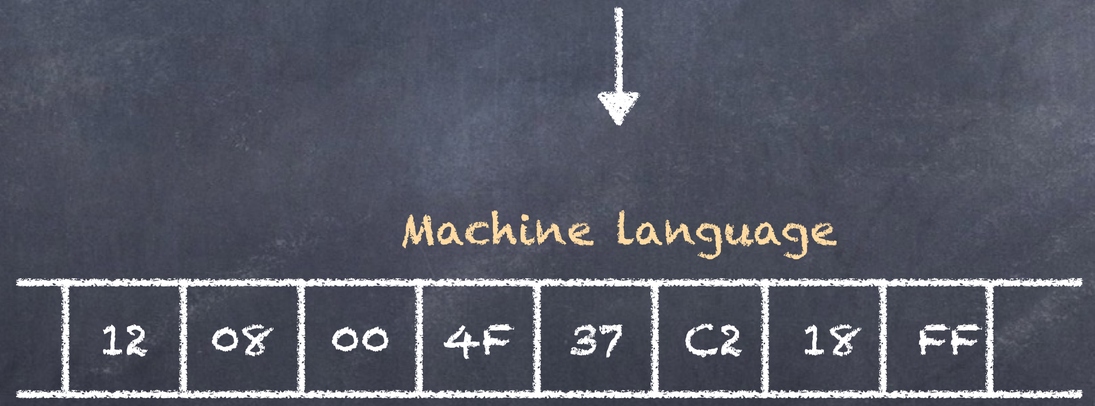
Intermediate language

Classical compiler

```
for (int i=0; i < list.Length; i++)
    result += i;
```



.NET compiler

```
for (int i=0; i < list.Length; i++)
    result += i;
```

Compilation

Intermediate language

```
ldc.i4.1
ldloc.1
add
stloc.0
```

JIT compilation

Machine language

12	08	00	4F	37	C2	18	FF	

Why 2 compilations?

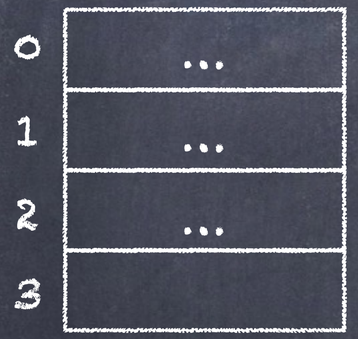
- JIT compiler can optimise for specific CPU
- Creates portable code that runs on any platform
- Code can be verified before running
- Code can be annotated with attributes

Disadvantages:

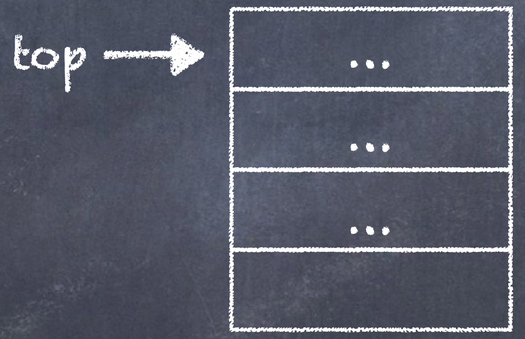
- Slightly slower than direct compilation

IL concepts

Local variable locations



Evaluation stack



IL instructions

```
ldc.i4.0
ldloc.1
add
stloc.1
ldloc.2
bne IL_0001F
ret
```

Code example

```
int i = 456;
i = i + 1;
```

ldc.i4.1

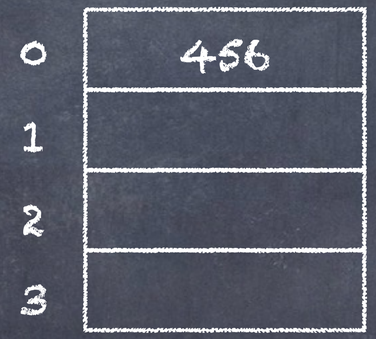
Load the 4-byte signed integer constant 1 on the evaluation stack.

ldloc.0

Load the variable in location 0 on the evaluation stack. The other value on the stack is pushed down.

add

Local variable locations



Evaluation stack

Add the top two numbers on the evaluation stack together, and replace them with the result

stloc.0

Remove the value at the top of the evaluation stack, and store it in location 0

Other instructions

- box: box the top value on the stack
- bne: branch if top 2 values on stack are not equal
- call: call a static member of a class
- callvirt: call an instance member of a class
- ldelem/stelem: load and store array elements
- newarr: create a new 1-dimensional array
- newobj: create a new object on the heap
- ret: return from method
- throw: throw an exception
- unbox: unbox the top reference on the stack

What have we learned?

- JIT compilation optimises for local hardware
- IL code is portable, can be verified and annotated
- IL uses local variable locations and an evaluation stack
- Built-in support for objects
- Built-in support for 1-dimensional arrays