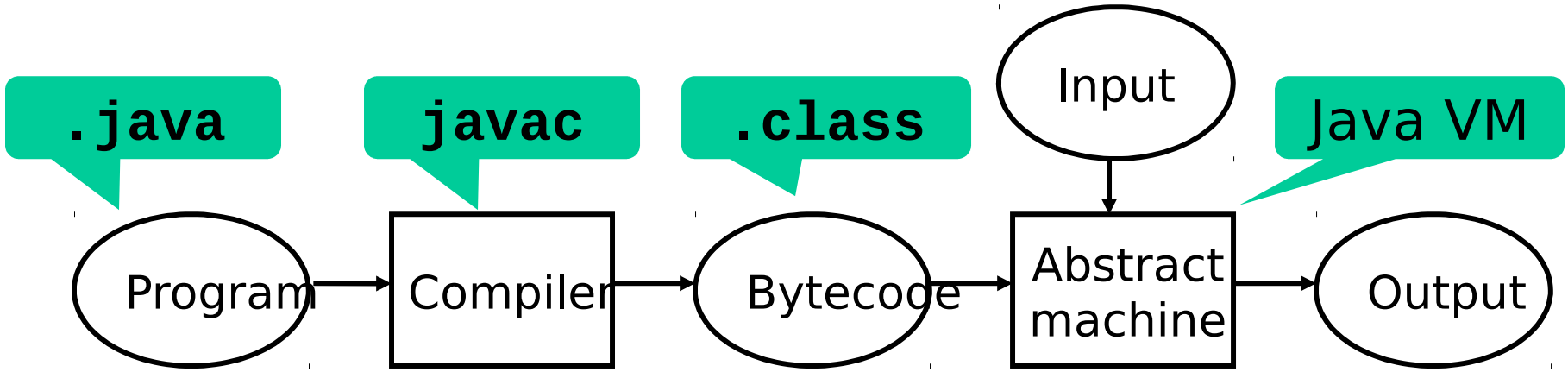
- Java Virtual Machine
- .NET Common Language Infrastructure (CLI)



Example Java program (ex6java.java)

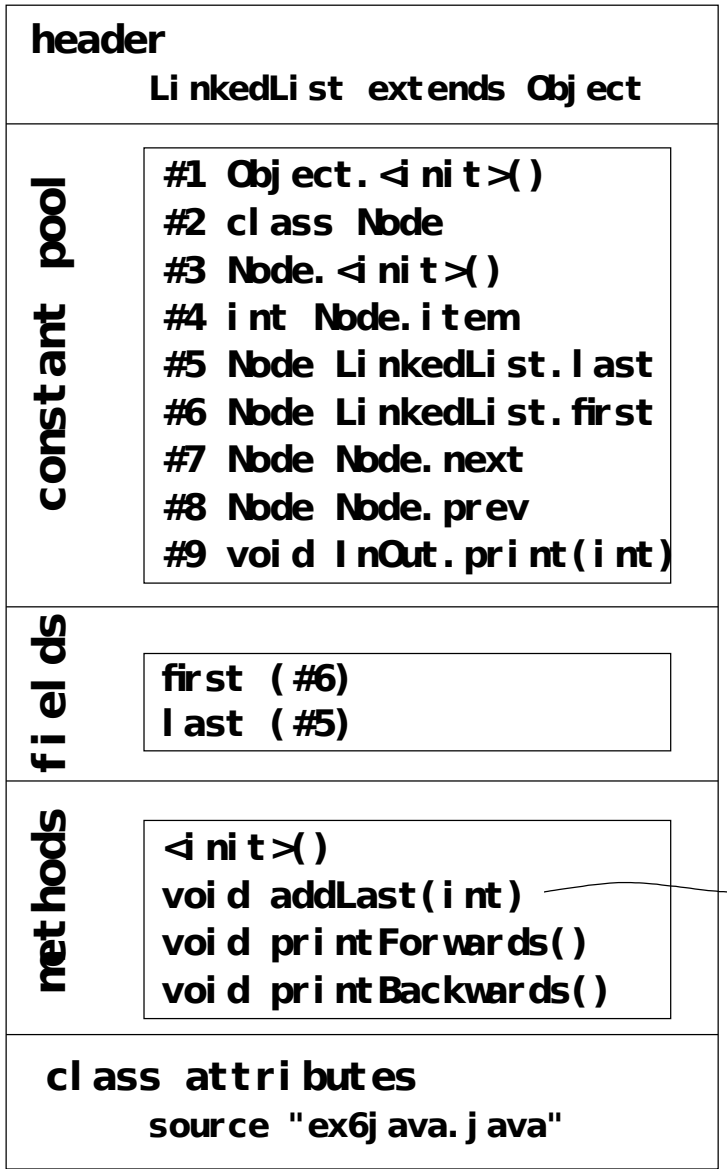
```
class Node extends Object {
  Node next;
  Node prev;
  int item;
}
```

```
class LinkedList extends Object {
  Node first, last;

  void addLast(int item) {
    Node node = new Node();
    node.item = item;
    if (this.last == null) {
      this.first = node;
      this.last = node;
    } else {
      this.last.next = node;
      node.prev = this.last;
      this.last = node;
    }
  }

  void printForwards() { ... }
  void printBackwards() { ... }
}
```

JVM class file (LinkedList.class)



Generated by
`javac ex6java.java`

Shown by
`javap -c -v LinkedList`

```
Stack=2, Locals=3, Args_size=2
 0 new #2 <Class Node>
 3 dup
 4 invokespecial #3 <Method Node()>
 7 astore_2
 8 aload_2
 9 iload_1
10 putfield #4 <Field int item>
13 ...
```

Some JVM bytecode instructions

Kind	Example instructions
push constant	iconst, ldc, aconst_null, ...
arithmetic	iadd, isub, imul, idiv, irem, ineg, iinc, fadd, ...
load local variable	iload, aload, fload, ...
store local variable	istore, astore, fstore, ...
load array element	iaload, baload, aaload, ...
stack manipulation	swap, pop, dup, dup_x1, dup_x2, ...
load field	getfield, getstatic
method call	invokestatic, invokevirtual, invokespecial
method return	return, ireturn, areturn, freturn, ...
unconditional jump	goto
conditional jump	ifeq, ifne, iflt, ifle, ...; if_icmpeq, if_icmpne, ...
object-related	new, instanceof, checkcast

Type prefixes: i=int, a=object, f=float, d=double, s=short, b=byte, ...

JVM bytecode verification

The JVM bytecode is *verified at load time,* before execution:

- An instruction must work on stack operands and local variables of the correct type
- A method must use no more local variables and no more local stack positions than it claims to
- For every point in the bytecode, the local stack has the same depth whenever that point is reached
- A method must throw no more exceptions than it admits to
- The execution of a method must end with a return or throw instruction, not `fall off the end'
- Execution must not use one half of a two-word value (e.g. a long) as a one-word value (int)

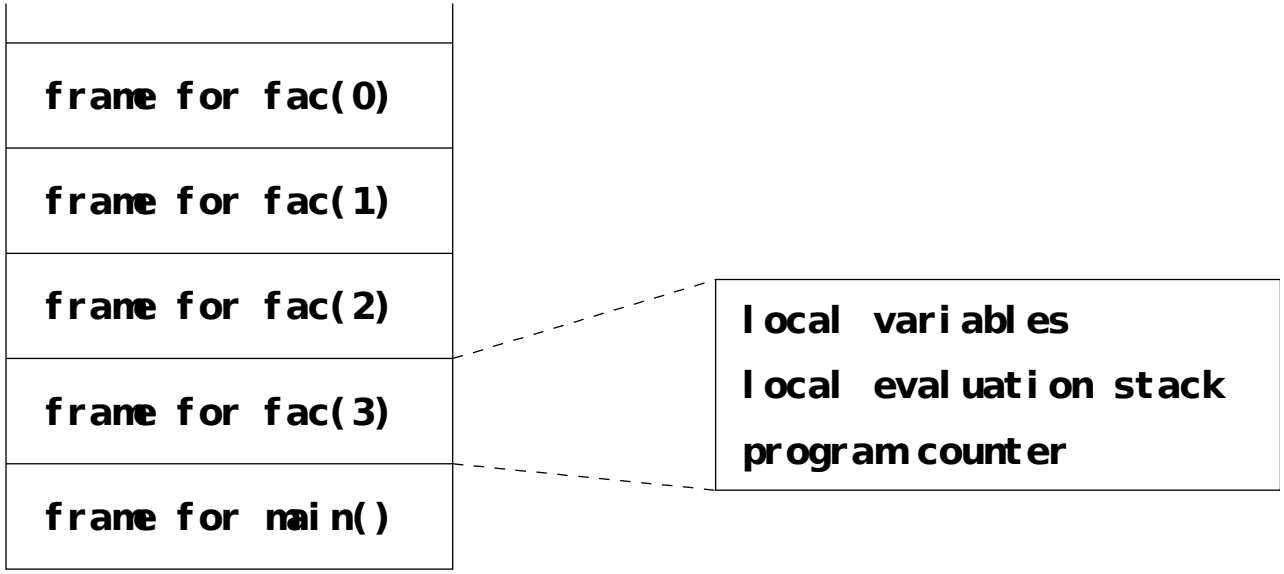
Additional JVM *runtime* checks

- Array-bounds checks on arr[i]
- Array assignment checks: Can store only subtypes of A into an A[] array
- Null-reference check (a reference is null *or* points to an object or array, because no pointer arithmetics)
- Checked casts: Cannot make arbitrary conversions between object classes
- Memory allocation succeeds or throws exception
- No manual memory deallocation or reuse

- Bottom line: A JVM program cannot read or overwrite arbitrary memory
- Better debugging, better security
- No buffer overflow attacks, worms, etc as in C/C++

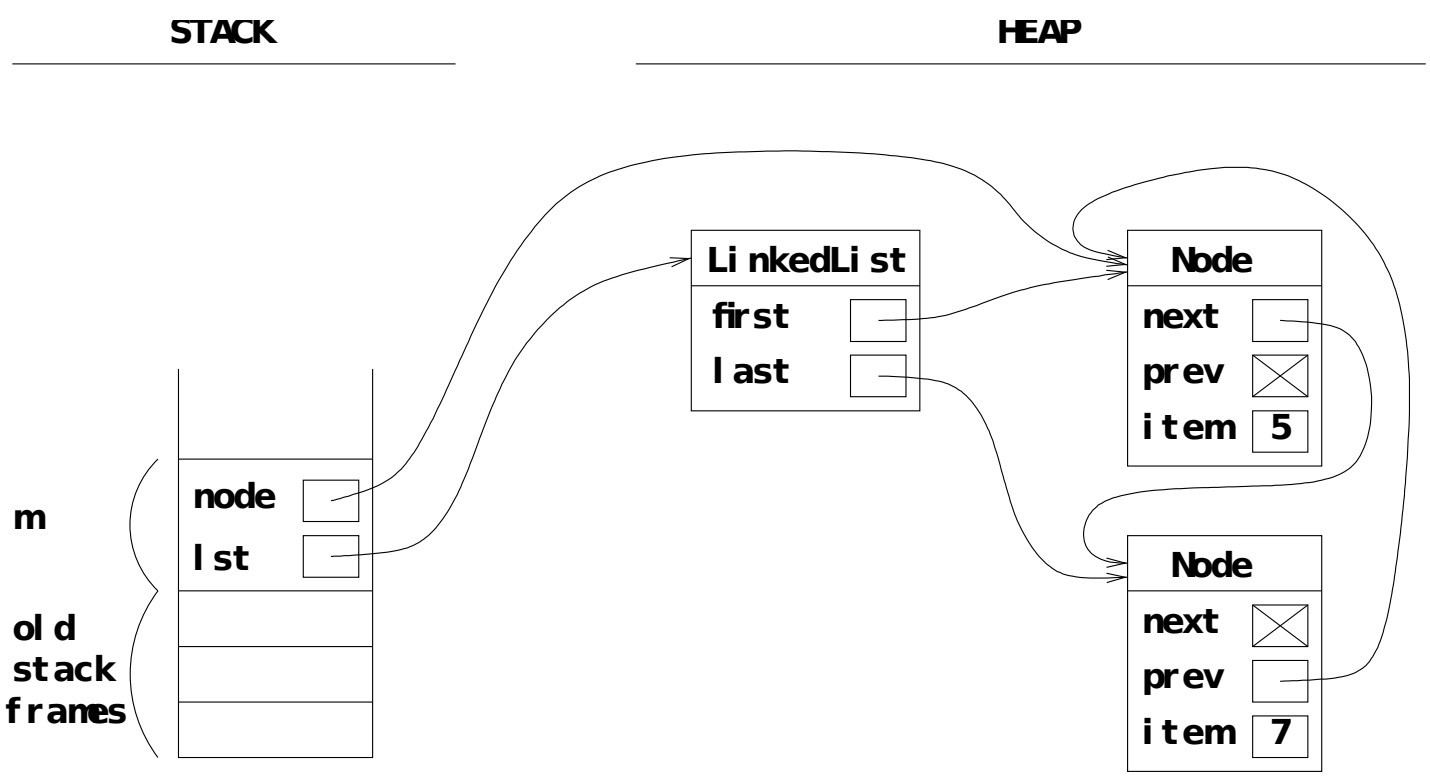
The JVM runtime stacks

- One runtime stack per thread
  - Contains activation records, one for each active function call
  - Each activation record has program counter, local variables, and local stack for intermediate results



Example JVM runtime state

```
void m() {
  LinkedList lst = new LinkedList();
  lst.addLast(5);
  lst.addLast(7);
  Node node = lst.first;
}
```



The .NET Common Language Infrastructure (CLI, CLR)

- Same philosophy and design as JVM
- Some improvements:
  - Standardized bytecode assembly (text) format
  - Better versioning, strongnames, …
  - Designed as target for multiple source languages (C#, VB.NET, JScript, Eiffel, F#, Python, Ruby, …)
  - User-defined value types (structs)
  - Tail calls to support functional languages
  - True generic types in bytecode: safer, more efficient, and more complex
- The .exe file = stub + bytecode
- Standardized as Ecma-335

Some .NET CLI bytecode instructions

Kind	Example instructions
push constant	ldc.i4, ldc.r8, ldnull, ldstr, ldtoken
arithmetic	add, sub, mul, div, rem, neg; add.ovf, sub.ovf, …
load local variable	ldloc, ldarg
store local variable	stloc, starg
load array element	ldelem.i1, ldelem.i2, ldelem.i4, ldelem.r8
stack manipulation	pop, dup
load field	ldfld, ldstfld
method call	call, calli, callvirt
method return	ret
unconditional jump	br
conditional jump	brfalse, brtrue; beq, bge, bgt, ble, blt, …; bge.un …
object-related	newobj, isinst, castclass

Type suffixes: i1=byte, i2=short, i4=int, i8=long, r4=float, r8=double, …

From Java and C# to bytecode

- Consider the Java/C#/C program ex13:

```
static void Main(string[] args) {
  int n = int.Parse(args[0]);
  int y;
  y = 1889;
  while (y < n) {
    y = y + 1;
    if (y % 4 == 0 && (y % 100 != 0 || y % 400 == 0))
      InOut.PrintI(y);
  }
  InOut.PrintC(10);
}
```

- Let us compile and disassemble it twice:
  - **javac ex13.java** then **javap –c ex13**
  - **csc /o ex13.cs** then **ildasm /text ex13.exe**

JVM

```
00 aload_0
01 iconst_0
02 aaload
03 invokestatic parseInt
06 istore_1
07 sipush 1889
10 istore_2
11 iload_2
12 iload_1
13 if_icmpge 48
16 iload_2
17 iconst_1
18 iadd
19 istore_2
20 iload_2
21 iconst_4
22 irem
23 ifne 11
26 iload_2
27 bipush 100
29 irem
30 ifne 41
33 iload_2
34 sipush 400
37 irem
38 ifne 11
41 iload_2
42 invokestatic printi
45 goto 11
48 bipush 10
50 invokestatic printc
53 return
```

.NET

```
0000 ldarg.0
0001 ldc.i4.0
0002 ldelem.ref
0003 call Parse
0008 stloc.0
0009 ldc.i4 0x761
000e stloc.1
000f br 003b
0014 ldloc.1
0015 ldc.i4.1
0016 add
0017 stloc.1
0018 ldloc.1
0019 ldc.i4.4
001a rem
001b brtrue 003b
0020 ldloc.1
0021 ldc.i4.s 100
0023 rem
0024 brtrue 0035
0029 ldloc.1
002a ldc.i4 0x190
002f rem
0030 brtrue 003b
0035 ldloc.1
0036 call PrintI
003b ldloc.1
003c ldloc.0
003d blt 0014
0042 ldc.i4.s 10
0044 call PrintC
0049 ret
```

.NET CLI has generic types, JVM doesn't

```
class CircularQueue<T> {
  private readonly T[] items;
  public CircularQueue(int capacity) {
    this.items = new T[capacity];
  }
  public T Dequeue() { ... }
  public void Enqueue(T x) { ... }
}
```

Source; generics

```
.class CircularQueue`1<T> ... {
  .field private initonly !T[] items
  ...
  .method !T  Dequeue() { ... }
  .method void Enqueue(!T x) { ... }
}
```

.NET CLI; generics

```
class CircularQueue ... {
  public java.lang.Object dequeue(); ...
  public void enqueue(java.lang.Object); ...
}
```

JVM; **no** generics

Consequences for Java

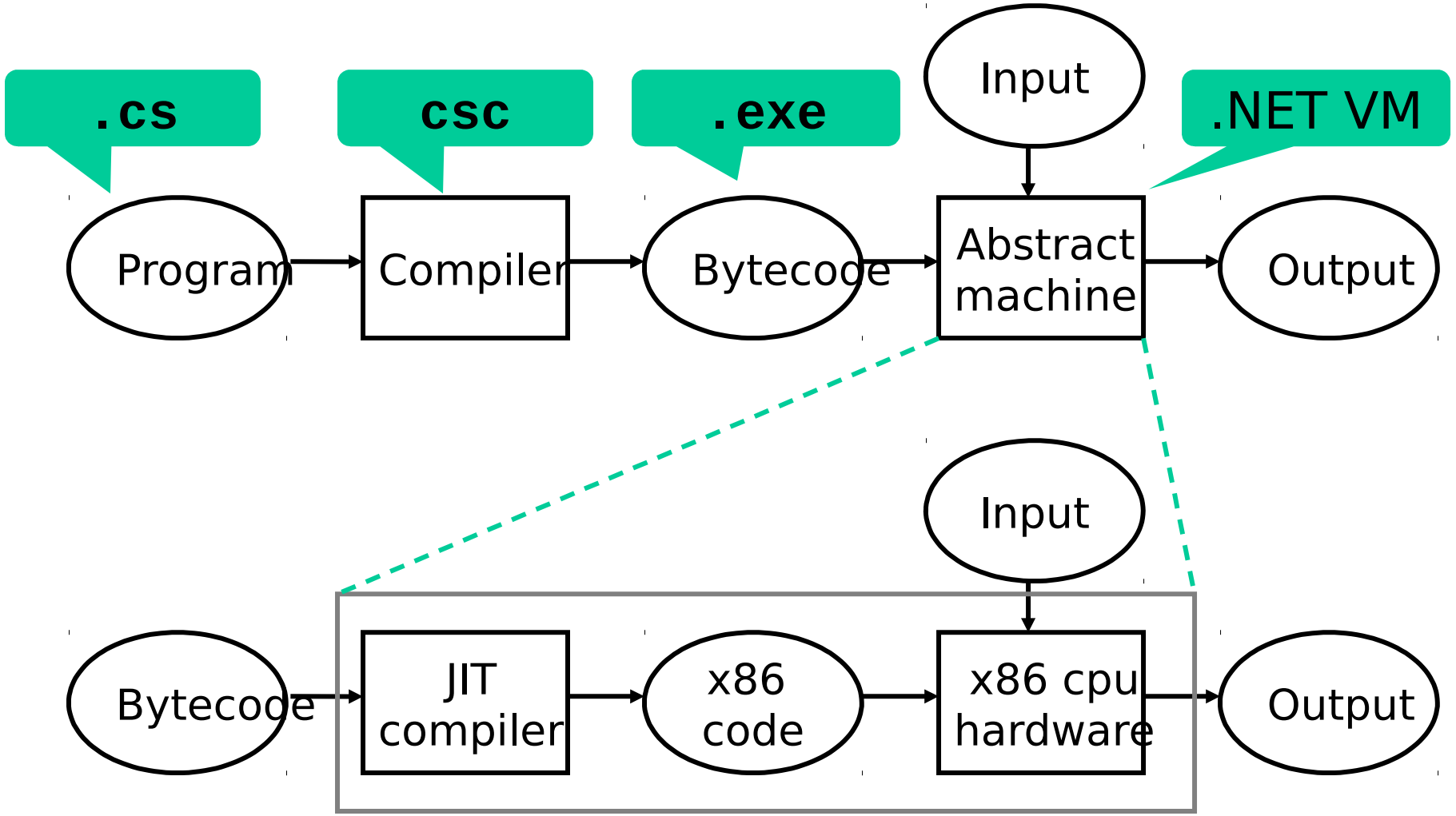
- The Java compiler replaces T
  - with **`Object`** in **`C<T>`**
  - with **`Mytype`** in **`C<T extends Mytype>`**
- So this **doesn't work** in Java, but works in C#:
  - Cast: **`(T)e`**
  - Instance check: **`(e instanceof T)`**
  - Reflection: **`T.class`**
  - Overload on different type instances of gen class:

```
void put(CircularQueue<Double> cqd) { ... }
void put(CircularQueue<Integer> cqd) { ... }
```

  - Array creation: **`arr = new T[10]`**
    So Java versions of **`CircularQueue<T>`** must use **`ArrayList<T>`**, not **`T[]`**

Just-in-time (JIT) compilation

- Bytecode is compiled to real (e.g. x86) machine code at runtime to get speed comparable to C/C++



Just-in-time compilation

- How to inspect .NET JITted code

```
static double Sqr(double x) {
  return x * x;
}
```

C#

```
csc /debug /o Square.cs
```

```
IL_0000:  ldarg.0
IL_0001:  ldarg.0
IL_0002:  mul
IL_0003:  ret
```

CLI

JIT compiler

```
00 pushl   %ebp
01 movl    %esp,%ebp
03 subl    $0x08,%esp
06 fldl    0x08(%ebp)
09 fldl    0x08(%ebp)
0c fmulp   %st,%st(1)
0e leave
0f ret
```

x86

```
movl   %ebp,%esp
popq   %ebp
```

Mono 3.2.3 MacOS 32 bit

```
mono –optimize=-inline
     –v –v Square.exe
```