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

#### header
- LinkedLIst extends Object

#### pool
- #1 Object.<init>()
- #2 class Node
- #3 Node.<init>()
- #4 int Node.item
- #5 Node.LinkedList.last
- #6 Node.LinkedList.first
- #7 Node.Node.next
- #8 Node.Node.prev
- #9 void InOut.print(int)

#### fields
- first (#6)
- last (#5)

#### methods
- <init>()
- void addLast(int)
- void printForwards()
- void printBackwards()

#### class attributes
- source "ex6java.java"

---

**Generated by**
- `javac ex6java.java`

**Shown by**
- `javap -c -v LinkedList`

---

---

**Stack=2, Local s=3, Args size=2**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>new #2 &lt;Class Node&gt;</td>
</tr>
<tr>
<td>3</td>
<td>dup</td>
</tr>
<tr>
<td>4</td>
<td>invokespecial #3 &lt;Method Node()&gt;</td>
</tr>
<tr>
<td>7</td>
<td>astore_2</td>
</tr>
<tr>
<td>8</td>
<td>aload_2</td>
</tr>
<tr>
<td>9</td>
<td>iload_1</td>
</tr>
<tr>
<td>10</td>
<td>putfield #4 &lt;Field int item&gt;</td>
</tr>
<tr>
<td>13</td>
<td>...</td>
</tr>
</tbody>
</table>
## Some JVM bytecode instructions

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

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

```
frame for fac(0)
frame for fac(1)
frame for fac(2)
frame for fac(3)
frame for main()
```

Local variables
Local evaluation stack
Program counter
Example JVM runtime state

```java
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

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

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:

```java
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
aload_0                | 0000  ldarg.0
iconst_0               | 0001  ldc.i4.0
aaload                  | 0002  ldelem.ref
invokestatic parseInt  | 0003  call Parse
istore_1                | 0008  stloc.0
sipush 1889             | 0009  ldc.i4 0x761
istore_2                | 000e  stloc.1
iload_2                 | 000f  br 003b
iload_1                 | 0014  ldloc.1
if_icmpge 48            | 0015  ldc.i4.1
iload_2                 | 0016  add
iconst_1                | 0017  stloc.1
iadd                     | 0018  ldloc.1
istore_2                | 0019  ldc.i4.4
iload_2                 | 001a  rem
iconst_4                | 001b  brtrue 003b
irem                    | 0020  ldloc.1
ifne 11                 | 0021  ldc.i4.s 100
iload_2                 | 0022  rem
bipush 100              | 0023  brtrue 0035
irem                    | 0024  ldloc.1
ifne 41                 | 0025  ldc.i4 0x190
iload_2                 | 0026  rem
sipush 400              | 002f  brtrue 003b
irem                    | 0030  ldloc.1
ifne 11                 | 0035  brtrue 003b
iload_2                 | 0036  call PrintI
invokestatic printi    | 003b  ldloc.1
goto 11                 | 003c  ldloc.0
goto 11                 | 003d  blt 0014
bipush 10               | 0042  ldc.i4.s 10
invokestatic printc     | 0044  call PrintC
return                  | 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) { ... }
}

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

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

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

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

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

```
csc /debug /o Square.cs
movl   %ebp,%esp
popq   %ebp
```

```
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
```

Mono 3.2.3 MacOS 32 bit

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

C# / CLI / x86

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

```
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
```

movl  %ebp,%esp
popq  %ebp