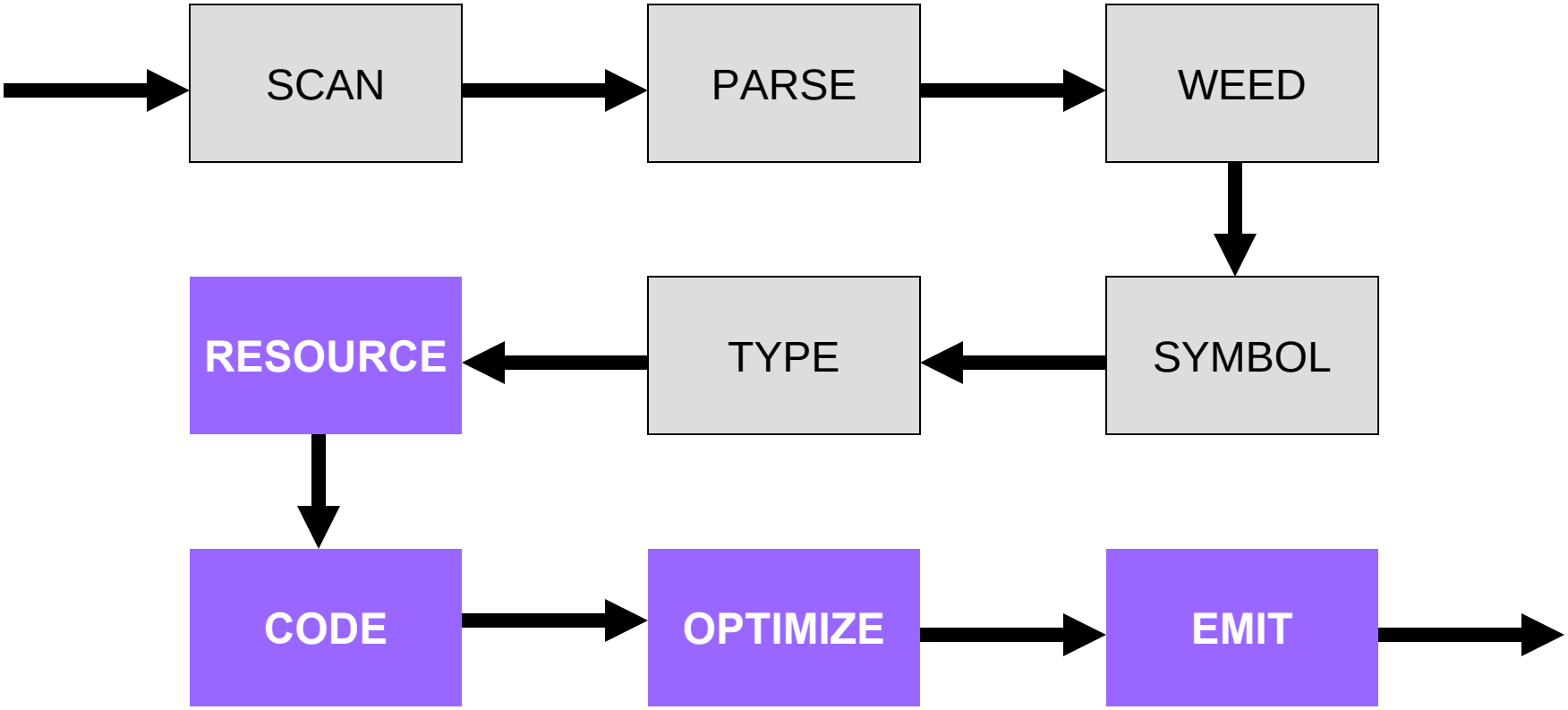
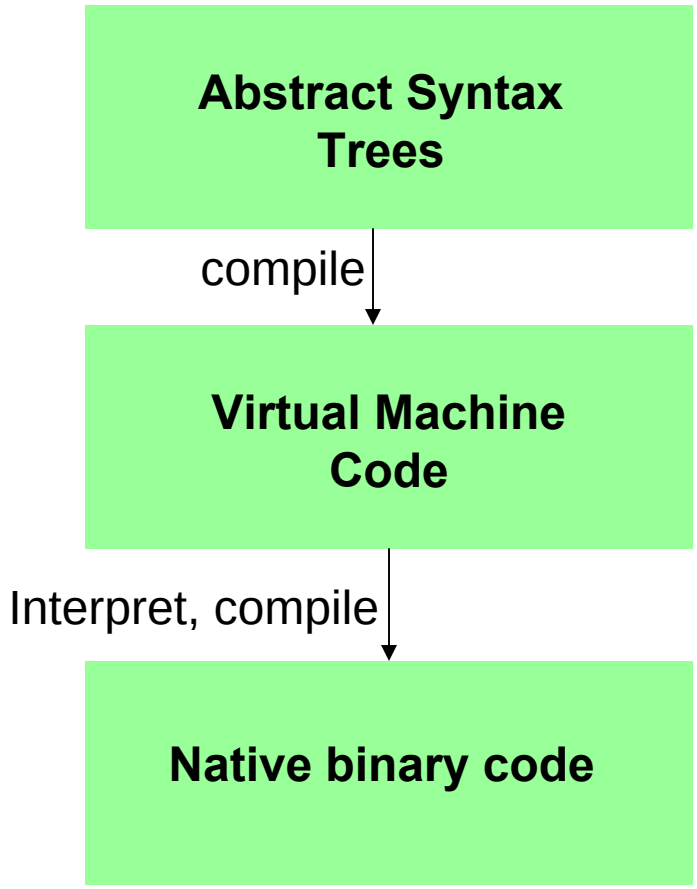
Compiler

Virtual Machines

Compiler Architecture



Virtual Machines



Pseudo-code (p-code), JVM

pentium, itanium

Compiling to VM Code

- Example:
  - gcc translates into Register Transfer Language (RTL), optimizes RTL, and then compiles RTL into native code.
- Advantages:
  - exposes many details of the underlying architecture; and
  - facilitates production of code generators for many target architectures.
- Disadvantage:
  - a code generator must be built for each target architecture.

Interpreting VM Code

- Examples:
  - P-code for early Pascal interpreters;
  - Postscript for display devices; and
  - Java bytecode for the Java Virtual Machine.
- Advantages:
  - easy to generate the code;
  - the code is architecture independent; and
  - bytecode can be more compact.
- Disadvantage:
  - poor performance due to interpretative overhead (typically 5-20 times slower).
    - advancements in Just In-Time (JIT) compiling closing the gap!

Java Virtual Machine

- memory;
- registers;
- condition codes; and
- execution unit.

JVM Memory

- a stack
  - used for function call frames;
- a heap
  - used for dynamically allocated memory;
- a constant pool
  - used for constant data that can be shared; and
- a code segment
  - used to store JVM instructions of currently loaded class files.

JVM Registers

- no general purpose registers;
- the stack pointer (`sp`) which points to the top of the stack;
- the local stack pointer (`lsp`) which points to a location in the current stack frame; and
- the program counter (`pc`) which points to the current instruction.

JVM Condition Codes

- stores the result of last instruction (in the stack) that can set condition codes (used for branching).

JVM Execution Unit

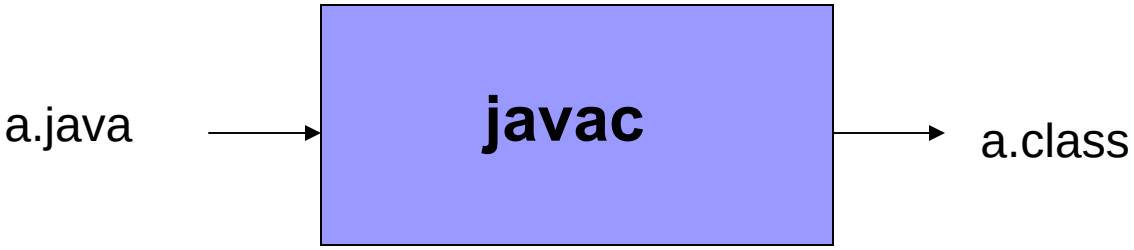
- reads the Java Virtual Machine instruction at the current pc, decodes the instruction and executes it;
- this may change the state of the machine (memory, registers, condition codes);
- the pc is automatically incremented after executing an instruction; but
- method calls and branches explicitly change the pc.

JVM Stack Frames

- Have space for:
  - a reference to the current object (`this`);
  - the method arguments;
  - the local variables; and
  - a local stack used for intermediate results.
- The number of local slots and the maximum size of the local stack are fixed at compile-time.

Java Compilation

- Java compilers translate source code to class files.
- Class files include the bytecode instructions for each method.




```
Magic number
Version number
Constant pool
Access flags
this class
super class
Interfaces
Fields
Methods
Attributes
```

Example Java Method

```
public class JVMExamples {
  public int abs(int x) {
    if (x < 0)
      return(x * -1);
    else
      return(x);
    }
}
```

Example JVM Bytecodes

```
public abs(I)I              // one int argument, returns an int
  L0 (0)                    // label L0
    LINENUMBER 3 L0         // line number 5
                            // --locals--  --stack---
    ILOAD 1                 // [ o -3 ]     [ -3  * ]
    IFGE L1                 // [ o -3 ]     [  *  * ]
  L2 (3)                    // label L2
    LINENUMBER 4 L2         // line number 6
    ILOAD 1                 // [ o -3 ]     [ -3  * ]
    ICONST_M1               // [ o -3 ]     [ -3 -1 ]
    IMUL                    // [ o -3 ]     [  3  * ]
    IRETURN                 // [ o -3 ]     [  *  * ]
  L1 (8)                    // label L1
    LINENUMBER 6 L1
    ILOAD 1
    IRETURN
  L3 (11)
    LOCALVARIABLE this LJVMExamples; L0 L3 0       // receiver object scope
    LOCALVARIABLE x I L0 L3 1     // scope of x
    MAXSTACK = 2                  // stack with 2 locations
    MAXLOCALS = 2                 // has space for 2 locals
```

Bytecode Interpreter

```
pc = code.start;
while(true) {
  npc = pc + inst_length(code[pc]);
  switch (opcode(code[pc])) {
    ILOAD_1: push(local[1]);
             break;
    ILOAD:   push(local[code[pc+1]]);
             break;
    ...
  }
  pc = npc;
}
```

Bytecode Interpreter

```
pc = code.start;
while(true){
  …
    ISTORE: t = pop();
            local[code[pc+1]] = t;
            break;
    IADD:   t1 = pop();  t2 = pop();
            push(t1 + t2);
            break;
    IFEQ:   t = pop();
            if (t == 0) npc = code[pc+1];
            break;
}
```

JVM Arithmetic Operators

```
ineg      [...:i]     -> [...:-i]
iadd      [...:i1:i2] -> [...:i1+i2]
isub      [...:i1:i2] -> [...:i1-i2]
imul      [...:i1:i2] -> [...:i1*i2]
idiv      [...:i1:i2] -> [...:i1/i2]
irem      [...:i1:t2] -> [...:i1\%i2]
iinc k a [...] -> [...]
          local[k]=local[k]+a
```

JVM Branch Operations

`goto L`	`[...] -> [...]` branch always
`ifeq L`	`[...:i] -> [...]` branch if `i == 0`
`ifne L`	`[...:i] -> [...]` branch if `i != 0`
`ifnull L`	`[...:o] -> [...]` branch if `o == null`
`ifnonnull L`	`[...:o] -> [...]` branch if `o != null`

More Branches

```
if_icmpeq L  [...:i1:i2] -> [...]
        branch if i1 == i2
if_icmpne L  [...:i1:i2] -> [...]
        branch if i1 != i2
if_icmpgt L  [...:i1:i2] -> [...]
        branch if i1 > i2
if_icmplt L  [...:i1:i2] -> [...]
        branch if i1 < i2
```

More Branches

```
if_icmple L  [...:i1:i2] -> [...]
        branch if i1 <= i2
if_icmpge L  [...:i1:i2] -> [...]
        branch if i1 >= i2
if_acmpeq L  [...:o1:o2] -> [...]
        branch if o1 == o2
if_acmpne L  [...:o1:o2] -> [...]
        branch if o1 != o2
```

Loading Constants

```
iconst_0      [...] -> [...:0]
iconst_1      [...] -> [...:1]
iconst_2      [...] -> [...:2]
iconst_3      [...] -> [...:3]
iconst_4      [...] -> [...:4]
iconst_5      [...] -> [...:5]
aconst_null   [...] -> [...:null]
ldc i         [...] -> [...:i]
ldc s         [...] -> [...:String(s)]
```

Memory Access

```
iload k              [...] -> [...:local[k]]
istore k             [...:i] -> [...]
                     local[k]=i
aload k              [...] -> [...:local[k]]
astore k             [...:o] -> [...]
                     local[k]=o
getfield f sig       [...:o] -> [...:o.f]
putfield f sig       [...:o:v] -> [...]
                     o.f=v
```

Stack Operations

```
dup     [...:v1] -> [...:v1:v1]
pop     [...:v1] -> [...]
swap    [...:v1:v2] -> [...:v2:v1]
nop     [...] -> [...]
```

Class Operations

```
new C  [...] -> [...:o]

instance_of C   [...:o] -> [...:i]
        if (o==null) i=0
        else i=(C<=type(o))

checkcast C  [...:o] -> [...:o]
        if (o!=null && !C<=type(o))
          throw ClassCastException
```

Method Operations

```
invokevirtual m sig [...:o:a_1:...:a_n] -> [...]

entry=lookup(m,sig,o.methods);
block=select(entry,type(o));
push frame of size block.locals+block.stacksize;
local[0]=o;
local[1]=a_1;
...
local[n]=a_n;
pc=block.code;
```

Method Operations

```
invokenonvirtual m sig
    [...:o:a_1:...:a_n] -> [...]

block=lookup(m,sig,o.methods);
push stack frame of size
  block.locals+block.stacksize;
local[0]=o;
local[1]=a_1;
...
local[n]=a_n;
pc=block.code;
```

Method Operations

```
ireturn    [...:i] -> [...]
           return i and pop stack frame

areturn    [...:o] -> [...]
           return o and pop stack frame

return     [...] -> [...]
           pop stack frame
```

A Java Method

```
class Cons {
  Object first;
  Cons rest;
  public boolean member(Object item) {
    if (first.equals(item))
      return true;
    else if (rest == null)
      return false;
    else
      return rest.member(item);
  }
}
```

Corresponding Bytecode

```
public member(Ljava/lang/Object;)Z // one Object argument, returns boolean
  L0 (0)                           // label L0
   LINENUMBER 5 L0                 // line number 5
   ALOAD 0                         // [ c o ] [ c * ]
   GETFIELD Cons.first : Ljava/lang/Object; // [ c o ] [ c.first * ]
   ALOAD 1                         // [ c o ] [ c.first o ]
   INVOKEVIRTUAL
     java/lang/Object.equals(Ljava/lang/Object;)Z // [ c o ] [ 1/0 * ]
   IFEQ L1                         // [ c o ] [ * * ]
  L2 (6)                           // label L2
   LINENUMBER 6 L2                 // line 37
   ICONST_1                        // [ c o ] [ 1 0 ]
   IRETURN                         // [ c o ] [ * * ]
  L1 (9)                           // label L1
   LINENUMBER 7 L1                 // line 7
   ALOAD 0                         // [ c o ] [ c * ]
   GETFIELD Cons.rest : LCons;     // [ c o ] [ c.rest * ]
   IFNONNULL L3                    // [ c o ] [ * * ]
  …
   MAXSTACK = 2
   MAXLOCALS = 2
```

Corresponding Bytecode

```
public member(Ljava/lang/Object;)Z
  …
  L4 (13)                              // label L4
    LINENUMBER 8 L4                    // line number 8
    ICONST_0                           // [ c o ] [ 0 * ]
    IRETURN                            // [ c o ] [ * * ]
  L3 (16)                              // label l3
    LINENUMBER 10 L3                   // line number 10
    ALOAD 0                            // [ c o ] [ c * ]
    GETFIELD Cons.rest : LCons;        // [ c o ] [ c.rest * ]
    ALOAD 1                            // [ c o ] [ c.rest o ]
    INVOKEVIRTUAL Cons.member(Ljava/lang/Object;)Z // [ c o ] [ 1/0 * ]
    IRETURN                            // [ c o ] [ * *]
  L5 (22)                              // label l5
    LOCALVARIABLE this LCons; L0 L5 0
    LOCALVARIABLE item Ljava/lang/Object; L0 L5 1
    MAXSTACK = 2
    MAXLOCALS = 2
```

Bytecode Verification

- bytecode cannot be trusted to be well-formed and well-behaved;
- before executing any bytecode that is received over the network, it should be verified;
- verification is performed partly at class loading time, and partly at run-time; and
- at load time, dataflow analysis is used to approximate the number and type of values in locals and on the stack.

Properties of Verified Bytecode

- each instruction must be executed with the correct number and types of arguments on the stack, and in locals (on all execution paths);
- at any program point, the stack is the same size along all execution paths; and
- no local variable can be accessed before it has been assigned a value.

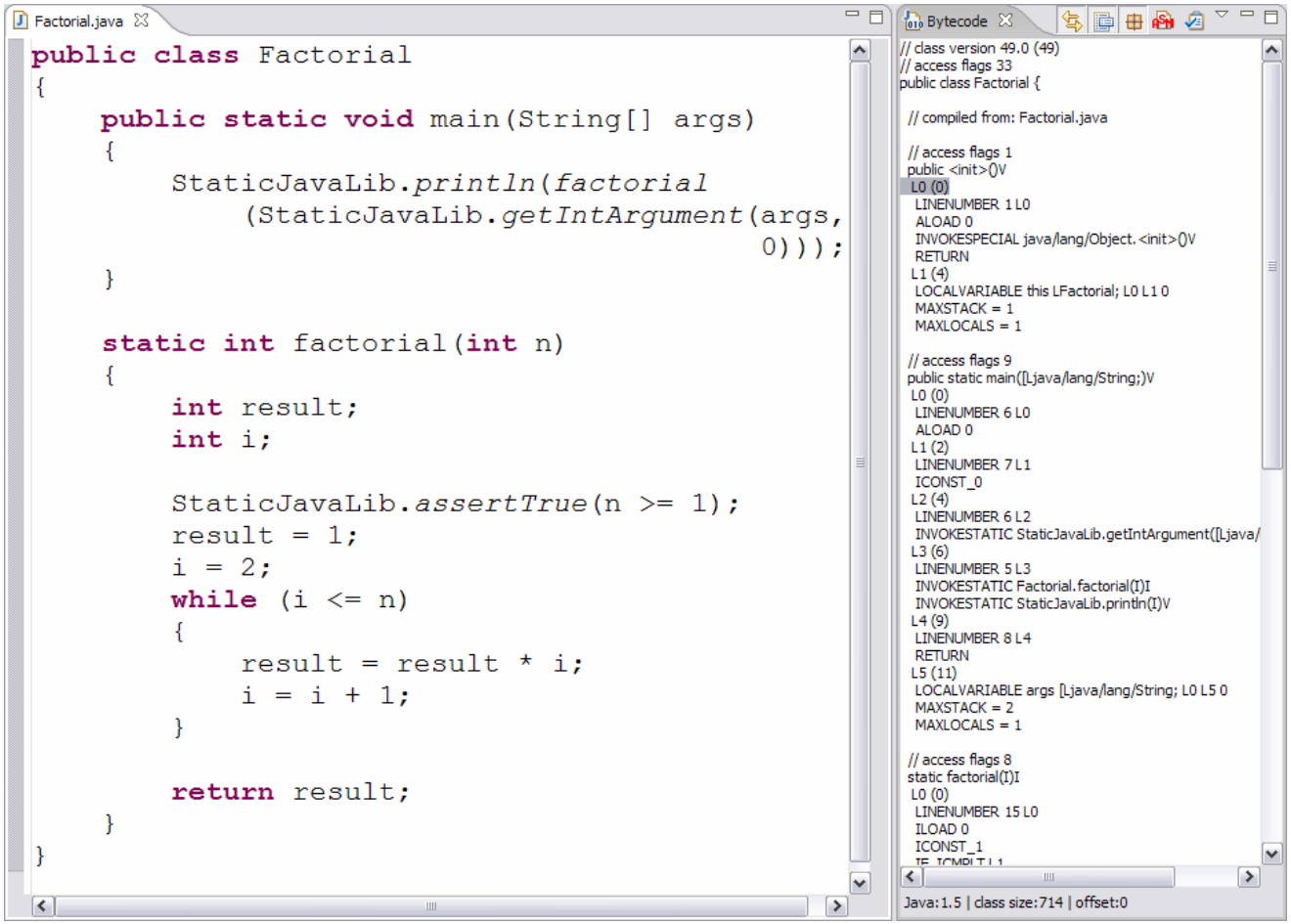
Interpreting Java

- when a method is invoked, a `ClassLoader` finds the correct class and checks that it contains an appropriate method;
- if the method has not yet been loaded, then it may be verified (remote classes);
- after loading and verification, the method body is interpreted; or
- the bytecode for the method is translated to native code (only for the first invocation).

ASM – A Java Bytecode Manipulation Framework

- can be used to
  - generate classes directly in binary form
  - modify classes
- similar tools/projects
  - BCEL
  - SERP
- advantages of ASM
  - small footprints
  - faster
  - has nice Eclipse plugins

ASM/Bytecode Outliner Plugins Demo



Creating Class using ASM

- ASM has utility classes/methods that makes it easier to write/read class files
  - we do not even need to know class file structure, or even maximum local variables and stack elements!
- Writing and reading class files are implemented using the Visitor pattern
- For more information
  - The ASM website contains excellent introductory tutorials on how to use ASM
  - ASM/Bytecode Outliner plugins have JVM instruction reference and the corresponding ASM visit methods

Creating Factorial Class using ASM

```
ClassWriter cw = new ClassWriter(false);

MethodVisitor mv;

cw.visit(V1_5, ACC_PUBLIC + ACC_SUPER,
         "Factorial", null, "java/lang/Object",
         null);
cw.visitSource("Factorial.java", null);
…
cw.visitEnd();

byte[] factorialClass = cw.toByteArray();
```

Creating Factorial's (Implicit) Constructor using ASM

```
public <init>()V

 L0 (0)
  LINENUMBER 1 L0
  ALOAD 0
  INVOKESPECIAL
   java/lang/Object…
  RETURN
 L1 (4)
  LOCALVARIABLE this
 LFactorial; L0 L1 0
    MAXSTACK = 1
    MAXLOCALS = 1
```

```
mv = cw.visitMethod(ACC_PUBLIC, "<init>",
                          "()V", null, null);
mv.visitCode();
Label l0 = new Label(); mv.visitLabel(l0);
mv.visitLineNumber(1, l0);
mv.visitVarInsn(ALOAD, 0);
mv.visitMethodInsn(INVOKESPECIAL,
        "java/lang/Object", "<init>", "()V");
mv.visitInsn(RETURN);
Label l1 = new Label(); mv.visitLabel(l1);
mv.visitLocalVariable("this", "LFactorial;",
                          null, l0, l1, 0);
mv.visitMaxs(1, 1);
mv.visitEnd();
```

Creating Factorial's `main` Method using ASM

```
public static
main([Ljava/lang/String;)V



L0 (0)
 LINENUMBER 6 L0
 ALOAD 0
L1 (2)
 LINENUMBER 7 L1
 ICONST_0
L2 (4)
 LINENUMBER 6 L2
 INVOKESTATIC
   StaticJavaLib…

L3 (6)
 LINENUMBER 5 L3
 INVOKESTATIC
   Factorial…
```

```
mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC,
                    "main",
                    "([Ljava/lang/String;)V",
                    null, null);
mv.visitCode();
Label l0 = new Label(); mv.visitLabel(l0);
mv.visitLineNumber(6, l0);
mv.visitVarInsn(ALOAD, 0);
Label l1 = new Label(); mv.visitLabel(l1);
mv.visitLineNumber(7, l1);
mv.visitInsn(ICONST_0);
Label l2 = new Label(); mv.visitLabel(l2);
mv.visitLineNumber(6, l2);
mv.visitMethodInsn(INVOKESTATIC, "StaticJavaLib",
                   "getIntArgument",
                   "([Ljava/lang/String;I)I");
Label l3 = new Label(); mv.visitLabel(l3);
mv.visitLineNumber(5, l3);
mv.visitMethodInsn(INVOKESTATIC, "Factorial",
                   "factorial", "(I)I");
```

Creating Factorial's `factorial` Method using ASM

```
static factorial(I)I


 L0 (0)
  LINENUMBER 15 L0
  ILOAD 0
  ICONST_1

  IF_ICMPLT L1
  ICONST_1

  GOTO L2
 L1 (6)
  ICONST_0
 L2 (8)
  INVOKESTATIC
    StaticJavaLib…
 L3 (10)
  LINENUMBER 16 L3
  ICONST_1
  ISTORE 1
```

```
mv = cw.visitMethod(ACC_STATIC, "factorial",
                    "(I)I", null, null);
mv.visitCode();
Label l0 = new Label(); mv.visitLabel(l0);
mv.visitLineNumber(15, l0);
mv.visitVarInsn(ILOAD, 0);
mv.visitInsn(ICONST_1);
Label l1 = new Label(); // create now, visit later
mv.visitJumpInsn(IF_ICMPLT, l1);
mv.visitInsn(ICONST_1);
Label l2 = new Label(); // create now, visit later
mv.visitJumpInsn(GOTO, l2);
mv.visitLabel(l1);      // visit L1 here
mv.visitInsn(ICONST_0);
mv.visitLabel(l2);      // visit L2 here
mv.visitMethodInsn(INVOKESTATIC, "StaticJavaLib",
                   "assertTrue", "(Z)V");
Label l3 = new Label(); mv.visitLabel(l3);
mv.visitLineNumber(16, l3);
mv.visitInsn(ICONST_1);
mv.visitVarInsn(ISTORE, 1);
```