CS577 W'04 Lecture Notes
Lecture 4

Example: Count

`Count.java`:

```
class Count {
  public static void main(String[] s) {
    int i;
    for (i = 0; i < 10; i++)
      System.out.println(i);
  }
}
```

```
% javac Count.java
% java Count
0
1
2
3
4
5
6
7
8
9
```


Example (cont.)

```
% javap -c Count
Compiled from Count.java
synchronized class Count extends java.lang.Object
    /* ACC_SUPER bit set */

    public static void main(java.lang.String[]);
    Count();


Method void main(java.lang.String[])
   0 iconst_0
   1 istore_1
   2 goto 15
   5 getstatic #6 <Field java.io.PrintStream out>
   8 iload_1
   9 invokevirtual #7 <Method void println(int)>
  12 iinc 1 1
  15 iload_1
  16 bipush 10
  18 if_icmplt 5
  21 return

Method Count()
   0 aload_0
   1 invokespecial #5 <Method java.lang.Object()>
   4 return
%
```


Java Virtual Machine Architecture

A JVM contains the following components:

Program Counter (per thread)

Stack (per thread)

Heap (shared) – contains all objects

Method Area (shared) – byte-codes and constant pools

Native method stacks (per thread, if required)

Method code is a sequence of **byte-code** instructions that implement methods (and constructors). The JVM byte-code is stack-based; most instructions take their operands from the stack and leave their results there.

Each class has a **constant pool**, which contains all the constant data referenced by the methods of that class, including numbers, strings, and symbolic names of other classes and members referenced by this class.

Stacks and Frames

There is one stack per thread. A stack consists of a sequence of **frames**; frames need not be contiguous in memory. Frame size and overall stack size may be limited by implementations.

One frame is associated with each method invocation. Each frame contains two areas, each of statically **fixed** size (per method):

• **local variable** storage associated with the method, and

• an **operand stack** for evaluating expressions within the method and for communicating arguments and results with other methods.

The local variable area is an array of words, addressed by word offset from the array base. Most locals occupy one word; long and double values occupy two consecutive words. The arguments to a method (including `this`, for instance methods) always appear as its initial local variables.

The operand stack is a stack of words. Most operands occupy one word; long and double values occupy two consecutive words, which must not be manipulated independently.

Frames may optionally contain additional information, e.g., for debugging.

Types and Verification

The JVM directly supports each of the primitive Java types (except `boolean`, which is mapped to `int`). Floating-point arithmetic follows IEEE 754. Values of reference types (classes,interfaces,arrays) are represented as heap pointers; layout of these values is implementation-dependent.

Data values are not tagged with type information, but instructions are. When executing, the JVM assumes that instructions are always operating on values of the correct type. The instruction set is designed to make it possible to **verify** that any given method is type-correct, without executing it. The JVM performs verification on any bytecode derived from an untrusted source (e.g., over the network).

At any given point of execution, each entry in the local variable area and the operand stack must have a well-defined **type state**; i.e., it must be possible to deduce the type of each entry unambiguously.

This is an unusual property for stacks! To enforce it, JVM code must be written with care. For example, when there are two execution paths to the same PC, they must arrive with identical type state. So, for example, it is impossible to to use a loop to copy an array onto the stack.

Instruction Set

Each JVM instruction consists of a one-byte **op code** followed by zero or more **parameters**. Instructions are only byte-aligned. Multi-byte parameters are stored in big-endian order.

The inner loop of the JVM execution engine (ignoring exceptions) is effectively:

```
do {
     fetch opcode;
     if (parameters) fetch parameters;
     execute action for opcode;
} while (more to do);
```

Most instructions take their operands from the top of the stack (popping them in the process) and push their result back on the top of the stack. A few operate directly on local variables.

Most instructions encode the type of their operands; thus, many instructions have multiple versions distinguished by their prefix (`i,l,f,d,b,s,c,a`).

The instruction set is not totally orthogonal; in particular, few operations are provided for bytes, shorts, and chars, and integer comparisons are much simpler than non-integer ones. In all, 201 out of 255 possible op-code values are used.

Families of instructions

Instructions group into families. Each family does the same basic operation, but has a variety of members distinguished by operand type and built-in arguments.

Example: `load` pushes the value of a local variable (specified as a parameter) onto the stack. Variants:

Load 1-word integer from local variable n:
`iload` n $(0 \leq n \leq 255)$
`iload_`n $(0 \leq n \leq 3)$
`wide iload` n $(0 \leq n \leq 65535)$

Load 2-word long from local variables n and $n+1$:
`lload` n $(0 \leq n \leq 255)$
`lload_`n $(0 \leq n \leq 3)$
`wide lload` n $(0 \leq n \leq 65535)$

Load 1-word float from local variables n:
`fload` n $(0 \leq n \leq 255)$
`fload_`n $(0 \leq n \leq 3)$
`wide fload` n $(0 \leq n \leq 65535)$

Load 2-word double from local variables n and $n+1$:
`dload` n $(0 \leq n \leq 255)$
`dload_`n $(0 \leq n \leq 3)$
`wide dload` n $(0 \leq n \leq 65535)$

Load 1-word object reference from local variable n:
`aload` n $(0 \leq n \leq 255)$
`aload_`n $(0 \leq n \leq 3)$
`wide aload` n $(0 \leq n \leq 65535)$

Load and Store

- `load` - push local variable onto stack
- `store` - pop top-of-stack into local variable
- `push`,`ldc`,`const` - push constant onto stack
- `wide` - modify following `load` or `store` to have wider parameter.

Arithmetic and Logic

- `add`,`sub`,`mul`, `div`, `rem`, `neg`
- `shl`,`shr`, `ushr`
- `or`, `and`, `xor`
- `iinc` - increment local variable

`div` and `rem` will throw an `ArithmeticException` given a zero divisor.

Conversions

- `i2l`,`i2f`,`i2d`,`l2f`,`l2d`,`f2d`.
- `i2b`,`i2c`,`i2s`, etc. - never raise exception.

Objects

- `new` – create new class instance
- `newarray` – creates new array
- `getfield`,`putfield` – access instance variables
- `getstatic`,`putstatic` – access class variables
- `aload`, `astore` – push, pop array elements to,from stack
- `arraylength`
- `instanceof`, `checkcast` – runtime narrowing checks

Stack management

- `pop`,`dup`,`dup_x`,`swap`

Control transfer

- `if_icmpeq`,`if_icmplt`, etc. – compare ints and branch
- `ifeq`,`iflt`, etc. – compare int with zero and branch
- `if_acmpeq`, `if_acmpne` – compare refs and branch
- `ifnull`,`ifnonnull` – compare ref with null and branch
- `cmp` – compare (non-integer) values and push result code (-1,0,1)
- `tableswitch`,`lookupswitch` – for switch statements
- `goto` – target is offset in method code
- `jsr`,`ret` – intended for `finally`
- `athrow` – throw explicit exception

Method invocation

- `invokevirtual` – for ordinary instance methods
- `invokeinterface` – for interface methods
- `invokespecial` – for constructor (`<init>`),`private`, or superclass methods
- `invokestatic` – for static methods
- `return`

Multiple Encodings

Some common operations can be implemented by more than one instruction, with differing levels of efficiency. For example, to load an integer constant i, we have:

One-byte sequences for $-1 \leq i \leq 5$

```
iconst_m1; iconst_0; iconst_1; iconst_2;
iconst_3; iconst_4; iconst_5
```

Two-byte sequences for $-128 \leq i \leq 127$

`bipush` i

Three-byte sequences for $-32768 \leq i \leq 32767$

`sipush` i

Two-byte sequences for arbitrary i loaded from first 255 entries in constant pool

`ldc` $< i >$

Three-byte sequences for arbitrary i loaded from any entry in constant pool

`ldc_w` $< i >$

`javac` should choose best available sequence based on i.

Constant Pool

The constant pool contains the following kinds of entries:

- `Utf8` – Unicode string in UTF-8 format.
- `Integer`,`Float`,`Long`,`Double`
- `String` – String, represented by `Utf8`
- `Class` – Fully-qualified Java class name, represented by `Utf8`
- `NameAndType` – Simple field or method name plus field or method **descriptor**, each represented by `Utf8`.
- `Fieldref`, `Methodref`, `InterfaceMethodref` – `Class` plus `NameAndType`.

Descriptors are strings that encode type information for fields or methods in terms of base types and fully-qualified class names. Method descriptors include the types of method parameters and result.

Java Class File Format

The class file format is the real standard of binary interoperability for JVM programs. Each class file describes a single class or interface. It is a stream of bytes, which may be obtained from a file, over a network, or elsewhere.

The class file contains:

- Magic number and compiler version information.
- Constant pool.
- Access flags for this class.
- Name of this class, its super-class, and its direct superinterfaces.
- Number, names, access flags, type descriptors, and values (if constant) for its fields.
- Number, names, access flags, type descriptors, code, and exception tables for its methods.
- Additional attribute information (e.g., for debugging) may be attached at the class, field, or method level.

Summary

JVM Bytecode is intended to be both easy to interpret and easy to use as compiler IR.

As an IR, it's pretty high-level.

Explicates:

- Parameter and local variable offsets
- Temporaries (using stack)
- Order of evaluation
- Control flow within procedures
- Exceptions

But NOT:

- Object layout and field offsets
- Method calls (virtual or otherwise)
- Inheritance hierarchy

Safety issues drove design

Interpreting Bytecode

```
/*
 * machine.c
 * Extremely simplified Java
 *   virtual machine interpreter.
 *
 *  Many things omitted, including:
 *   - synchronization for threads
 *   - exception handling
 *   - wide arguments
 *   - multiple types of data
 *
 * Derived from kaffe by Tim Wilkinson
 * Copyright (c) 1996
 * T. J. Wilkinson & Associates, London, UK.
 */

typedef unsigned char bytecode;
#define NOP 0
#define ACONST_NULL 1
#define ICONST_M1 2
...

const uint8 insnLen[256] =
        1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        ...
```



```
void
virtualMachine(methods* meth, int* args, int* retval)
{
  /* If these can be kept in registers then things
   *   will go much faster.
   */
  register bytecode* code;  /* code array */
  register int* lcl;        /* operand stack */
  register int* sp;         /* operand stack pointer
  register uintp pc = 0;    /* current code pointer *
  register uintp npc = 0;   /* next code pointer * /

  /* Allocate stack space and locals. */
  lcl = alloca(sizeof(int) *
               (meth->localsz + meth->stacksz));

  /* Determine number of arguments */
  nargs = meth->...;

  /* Copy in the arguments */
  sp = lcl;
  args = &args[nargs-1];
  for (i = 0; i < nargs; i++)
    *(sp++) = *(nargs--);

  sp = &lcl[meth->localsz + meth->stacksz];

  code = (bytecode*)meth->c.bcode.code;
```

```
/* Execute the code */
for (;;) {
  pc = npc;
  npc = pc + insnLen[code[pc]];
  switch(code[pc]) {

  NOP:
    break;

  ACONST_NULL:
    *(--sp) = 0;
    break;

  ...

  BIPUSH:
    *(--sp) = (int8)code[pc+1];
    break;

  ...

  ILOAD:
    idx = (uint8)code[pc+1];
    *(--sp) = *(lcl+idx)
    break;

  ...
```



```
  ISTORE:
    idx = (uint8)code[pc+1];
    *(lcl+idx) = *(sp++)
    break;

  ...

  DUP_X1:
    sp--;
    *sp = *(sp+1);
    *(sp+1) = *(sp+2);
    *(sp+2) = *sp;
    break;

  ...

  IADD:
    *(++sp) = *sp + *(sp+1)
    break;

  ...

  IINC:
    idx = (uint8)code[pc+1];
    *(lcl+idx) = *(lcl+idx) + (int8)code[pc+2];
    break;

  ...
```



```
  IFEQ:
    idx = (int16)((code[pc+1] << 8) | code[pc+2]);
    if (*sp++ == 0) npc = pc+idx;
    break;
  ...

  GETFIELD:
    idx = (uint16)((pc[1] << 8) | pc[2]);
    offset = get_field_offset(idx); /* some magic *
    *sp = *(*sp+offset);
    break;

  ...

  INVOKESTATIC:
    idx = (uint16)((pc[1] << 8) | pc[2]);
    method = get_method_info(idx); /* magic */
    nargs = method->...;
    virtualMachine(method, sp, retval);
    sp += (nargs -1);
    *sp = *retval;
    break;

  IRETURN:
    *retval = *sp;
    goto end;

  ...
  }
 }
 end:
}
```


Why are Interpreters Slow?

Interpreters are (relatively) simple to write, (usually) portable, and offer fast turn-around during development.

But they are slow! Why?

Elements of cost per (virtual) instruction:

- Dispatch (fetch,code and start).
- Access arguments.
- Perform function (usually cheap!).

Target architecture very important (even if not directly exposed).

- Register-rich?
- Memory hierarchy? Costs of using underlying memory-based stack?
- Indirect jump support?

Usual tradeoff between speed and portability.

Making Dispatch Cheaper

• Threaded code: implementation of each instruction concludes by jumping to code for next instruction (avoiding jump to `case`).

• Represent each instruction by address of its implementation (avoiding jump table lookup).

• Build combined instuctions (decrease instruction count; increase amount of useful work per decoded instruction).

• Use stack architecture: no explicit arguments, so no decoding cost. (But may require more instructions; some recent debate on this.)

• Ultimately: inline code sequences (avoiding dispatch altogether) – really a (very) simple JIT compiler.

• Optimize to make better use of hardware branch prediction (see Ertl and Gregg paper).

Making Argument Access Cheaper

• Build specialized instructions based on common literal arguments (so no need to access arguments).

• Attempt **stack caching** (static or dynamic): hold top VM stack elements in (real) registers rather than memory.

• (Note: using a register VM architecture doesn't make it easier to hold data in real registers!)

General Challenges: Portability, avoiding possible code explosion.