Why study the Java Virtual Machine (JVM)?

• Understanding (and improving on) the output of Java compilers (just like looking at machine code from C).

• Understanding the input to just-in-time compilers and other optimized Java execution engines.

• Understanding portability issues.

• Understanding security issues.

• JVM can be used as a target for other languages.

• You might possibly want to implement your own JVM.

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.

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.

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.

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

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.

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`

Resolution

Entries in the constant pool are **resolved** when first referenced by an executing instruction. Resolution of `ref` constants is a very complex process involving **loading**, **linking** (**verifying** and **preparing**), and **initializing** the `Class` of the `ref` and any classes on which it depends.

The end result of the resolution process is a direct pointer to the runtime representation of an object and/or an offset into that representation. This information is used to execute the provoking instruction, and may be used to rewrite that instruction into a more efficient form (e.g., Suns' `quick` form instructions).

Once a constant pool entry has been resolved, subsequent references to it always use the results of the original resolution.

Exception table

In addition to a sequence of byte-code instructions, each method has an **exception table** describing the all the exception handlers defined for the method.

Each entry in the table describes one handler body, and gives:

- The starting and ending PCs for which the handler applies.
- The PC of the handler code.
- The subclass of `Throwable` caught by this exception.

PC ranges are always either nested or discrete, never overlapping.

When an exception is raised, the table is searched in order; the first entry whose PC range covers the raise PC and whose class matches the thrown value is invoked.

Code for handlers is normally just appended to the main-line method code.

Example: Manipulating Doubles

```
public static void main(String[] s) {
  double d;
  for (d = 0; d < 10.0; d++)
    System.out.println(d);
}

Method void main(java.lang.String[])
   0 dconst_0
   1 dstore_1
   2 goto 16
   5 getstatic #6 <Field java.io.PrintStream out>
   8 dload_1
   9 invokevirtual #7 <Method void println(double)>
  12 dload_1
  13 dconst_1
  14 dadd
  15 dstore_1
  16 dload_1
  17 ldc2_w #8 <Double 10.0>
  20 dcmpg
  21 iflt 5
  24 return
```

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.

Example: Trivial Arithmetic

```
  static boolean right(int a, int b, int c) {
        return (a*a + b*b) == (c*c);
  }

Method boolean right(int, int, int)
   0 iload_0
   1 iload_0
   2 imul
   3 iload_1
   4 iload_1
   5 imul
   6 iadd
   7 iload_2
   8 iload_2
   9 imul
  10 if_icmpeq 15
  13 iconst_0
  14 ireturn
  15 iconst_1
  16 ireturn
```

Example: More Arithmetic

```
  static double quad(double a, double b, double c) {
    return (-b + java.lang.Math.sqrt(b*b - 4*a*c))
           / (2 * a);
  }

Method double quad(double, double, double)
   0 dload_2
   1 dneg
   2 dload_2
   3 dload_2
   4 dmul
   5 ldc2_w #8 <Double 4.0>
   8 dload_0
   9 dmul
  10 dload 4
  12 dmul
  13 dsub
  14 invokestatic #5 <Method double sqrt(double)>
  17 dadd
  18 ldc2_w #6 <Double 2.0>
  21 dload_0
  22 dmul
  23 ddiv
  24 dreturn
```

Example: Operand Stack Manipulation (2)

```
class Next {
  private long index = 0;
  public long nextIndex() {
        long j = index;
        index++;
        return j;
  }
}

Method long nextIndex()
   0 aload_0
   1 getfield #4 <Field long index>
   4 lstore_1
   5 aload_0
   6 dup
   7 getfield #4 <Field long index>
  10 lconst_1
  11 ladd
  12 putfield #4 <Field long index>
  15 lload_1
  16 lreturn
```

Example: Operand Stack Manipulation (1)

```
class Next {
  private long index = 0;
  public long nextIndex() {
    return index++;
  }
}

Method long nextIndex()
   0 aload_0
   1 dup
   2 getfield #4 <Field long index>
   5 dup2_x1
   6 lconst_1
   7 ladd
   8 putfield #4 <Field long index>
  11 lreturn
```

Example: Arrays

```
double dot(double[] a,double[] b) {
  int i;
  double c = 0.0;
  for (i = 0; i < a.length; i++)
    c += a[i] * b[i];
  return c;
}

Method double dot(double[], double[])
   0 dconst_0
   1 dstore 4
   3 iconst_0
   4 istore_3
   5 goto 23
   8 dload 4
  10 aload_1
  11 iload_3
  12 daload
  13 aload_2
  14 iload_3
  15 daload
  16 dmul
  17 dadd
  18 dstore 4
  20 iinc 3 1
  23 iload_3
  24 aload_1
  25 arraylength
  26 if_icmplt 8
  29 dload 4
  31 dreturn
```

Example: Objects

```
class MyObj {
  int i;
  MyObj(int j) {i = j;}
  static MyObj mkMyObj(int k) {
    return new MyObj(k + k);
  }
  int getMyObj() {return i;}
  static int mkgetMyObj() {
    MyObj m = mkMyObj(10);
    return m.getMyObj();
  }
}

Method MyObj(int)
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 aload_0
   5 iload_1
   6 putfield #6 <Field int i>
   9 return

Method MyObj mkMyObj(int)
   0 new #1 <Class MyObj>
   3 dup
   4 iload_0
   5 iload_0
   6 iadd
   7 invokespecial #4 <Method MyObj(int)>
  10 areturn
```

Example: Objects (2)

```
class MyObj {
  int i;
  MyObj(int j) {i = j;}
  static MyObj mkMyObj(int k) {
    return new MyObj(k + k);
  }
  int getMyObj() {return i;}
  static int mkgetMyObj() {
    MyObj m = mkMyObj(10);
    return m.getMyObj();
  }
}

Method int getMyObj()
   0 aload_0
   1 getfield #6 <Field int i>
   4 ireturn

Method int mkgetMyObj()
   0 bipush 10
   2 invokestatic #7 <Method MyObj mkMyObj(int)>
   5 astore_0
   6 aload_0
   7 invokevirtual #5 <Method int getMyObj()>
  10 ireturn
```

Example: Objects(3)

```
class YourObj extends MyObj {
  YourObj(int j) {super(j*j);}
  int getYourObj() {return -i;}
  static void funWithYourObj() {
    YourObj y = new YourObj(10);
    MyObj m = y;
    y.getYourObj();
    y.getMyObj();
    m.getMyObj();
    y = (YourObj) m;
  }
}

Method YourObj(int)
   0 aload_0
   1 iload_1
   2 iload_1
   3 imul
   4 invokespecial #3 <Method MyObj(int)>
   7 return

Method int getYourObj()
   0 aload_0
   1 getfield #7 <Field int i>
   4 ineg
   5 ireturn
```

Example: Objects(4)

```
class YourObj extends MyObj {
  ...
  static void funWithYourObj() {
    YourObj y = new YourObj(10);
    MyObj m = y;
    y.getYourObj();
    y.getMyObj();
    m.getMyObj();
    y = (YourObj) m;
  }
}

Method void funWithYourObj()
   0 new #2 <Class YourObj>
   3 dup
   4 bipush 10
   6 invokespecial #4 <Method YourObj(int)>
   9 astore_0
  10 aload_0
  11 astore_1
  12 aload_0
  13 invokevirtual #6 <Method int getYourObj()>
  16 pop
  17 aload_0
  18 invokevirtual #5 <Method int getMyObj()>
  21 pop
  22 aload_1
  23 invokevirtual #5 <Method int getMyObj()>
  26 pop
  27 aload_1
  28 checkcast #2 <Class YourObj>
  31 astore_0
  32 return
```