VIRTUAL MACHINES

- Widely used at both language and whole-system level.
- Offer enhanced portability, by abstracting away from specifics of underlying target platform.
- VM code is a well-specified intermediate representation that can be processed in many useful ways:
  - transmitted
  - interpreted
  - compiled
  - linked
  - verified
  - ...

JAVA ARCHITECTURE

- Mandated separation of front end and back end with precisely specified intermediate code.
- Back end doesn’t trust provider of byte codes; hence verification step in JVM.
- Focus on high-speed compilation:
  - JIT compilers
  - mixed interpreter/compiler (eg HotSpot)
  - feedback-directed optimization
- Focus on resource-bounded compilation and execution environment.
- Dynamic loading (and even reloading) of class definitions.
- Microsoft’s version of picture explicitly makes byte code (CIL) a multi-language common ground.
JAVA ARCHITECTURE ISSUES

• Except for the need to support dynamic loading, we could dispense with byte code and JVM, and use standard compiler architecture for Java too; some experimental systems do.

• Byte code is relatively high-level for IR (can recover source from it), and is better suited to being interpreted than to being optimized, so compiler in JVM often uses lower-level IR.

• We can essentially dispense with front-end and just treat byte-code as source.

JAVA EXAMPLE: SOURCE CODE

Count.java:

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

JAVA EXAMPLE: COMPILING AND EXECUTION

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

BYTECODE FOR COUNT

% javap -c Count
Compiled from "Count.java"
class Count extends java.lang.Object{
   count();
   Code:
      0 : aload_0
      1 : invokespecial #1; //Method java/lang/Object."<init>"();V
      4 : return

   public static void main(java.lang.String[]);
   Code:
      0 : iconst_0
      1 : istore_1
      2 : iload_1
      3 : bipush 10
      4 : if_icmpge 21
      5 : getstatic #2; //Field java/lang/System.out:Ljava/io/PrintStream;
      8 : iload_1
      9 : invokevirtual #3; //Method java/io/PrintStream.println:(I)V
     12 : iinc 1, 1
     15 : goto 2
     18 : goto 2
     21 : return
}
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.

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.

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 (there are 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.
INSTRUCTION SET ORGANIZATION

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).

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.

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 OPERATIONS (1)

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

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

EXAMPLE FAMILY

Load 1-word integer from local variable n:
ilo d n (0 ≤ n ≤ 255)
ilo ad n (0 ≤ n ≤ 3)
wide il oad n (0 ≤ n ≤ 65535)

Load 2-word long from local variables n and n + 1:
ilo ad n (0 ≤ n ≤ 255)
ilo ad n (0 ≤ n ≤ 3)
wide iload n (0 ≤ n ≤ 65535)

Load 1-word float from local variables n:
float n (0 ≤ n ≤ 255)
float ad n (0 ≤ n ≤ 3)
wide float ad n (0 ≤ n ≤ 65535)

Load 2-word double from local variables n and n + 1:
dload n (0 ≤ n ≤ 255)
dload ad n (0 ≤ n ≤ 3)
wide dload n (0 ≤ n ≤ 65535)

Load 1-word object reference from local variable n:
aload n (0 ≤ n ≤ 255)
aload ad n (0 ≤ n ≤ 3)
wide aload n (0 ≤ n ≤ 65535)

STACK MANAGEMENT

- push, dup, dup x, swap

Control transfer
- if_icmp eq, if_icmp l t, etc. - compare ints and branch
- ifeq, ifil t, etc. - compare int with zero and branch
- if_acmp eq, if_acmp ne - compare refs and branch
- ifnonnull, ifnonnull - compare ref with null and branch
- cmp - compare (non-integer) values and push result code (-1, 0, 1)
- tables w itch, lookupswitch - for switch statements
- goto - target is offset in method code
- jsr, ret - intended for finally
- athrow - throw explicit exception

MORE OPERATIONS (2)
**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

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

**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.

**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 ≤ i ≤ 5`**
- `iconst_m1`
- `iconst_0`
- `iconst_1`
- `iconst_2`
- `iconst_3`
- `iconst_4`
- `iconst_5`

**Two-byte sequences for `-128 ≤ i ≤ 127`**
- `bipush i`

**Three-byte sequences for `-32768 ≤ i ≤ 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`

**Constant Pool Example: Count**

```java
% javac -v Count
Constant pool:
const #0 = Method #5.#14; // java/lang/Object."<init>":()V
const #1 = Method #5.#14; // java/lang/Object."<init>":()V
const #2 = Field #15.#16; // java/lang/System.out:Ljava/io/PrintStream;
const #3 = Method #17.#18; // java/io/PrintStream.println:(I)V
const #4 = class #19; // Count
const #5 = class #20; // java/lang/Object
const #6 = Asciz <init>;
const #7 = Asciz ()V;
const #8 = Asciz Code;
const #9 = Asciz LineNumberTable;
const #10 = Asciz main;
const #11 = Asciz java/lang/String;V;
const #12 = Asciz SourceFile;
const #13 = Asciz Count.java;
const #14 = NameAndType #6:#7; // "<init>":(V
const #15 = class #21; // java/lang/System
const #16 = NameAndType #22:#23; // out:Ljava/io/PrintStream;
const #17 = class #24; // java/io/PrintStream
const #18 = NameAndType #25:#26; // println:(I)V
const #19 = Asciz Count;
const #20 = Asciz java/lang/Object;
const #21 = Asciz java/lang/System;
const #22 = Asciz out;
const #23 = Asciz java/io/PrintStream;
const #24 = Asciz java/io/PrintStream;
const #25 = Asciz println;
const #26 = Asciz (I)V;
```
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.

JVM Bytecode is intended to be both easy to interpret and easy to use as compiler IR. As an IR, it’s fairly high-level (largely for safety reasons). It makes the following explicit:

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

But it leaves the following implicit:

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

All these must be resolved inside the JVM implementation.