Java Virtual Machine

Part 2: the JBC & class files

Java Byte Code

- As assembly language code is to most HLLs, JBC is to Java
- Although most programmers work at the high level, a thorough understanding of the lower level helps us achieve better performing, lower cost software

JVM instructions*

- A JVM instruction consists of a one-byte opcode specifying the operation to be performed, followed by zero or more operands supplying arguments or data that are used by the operation
- Many instructions have no operands and consist only of an opcode

* This and the next several slides are almost verbatim from the official reference on all things JVM: the quoted parts are in purple:

The JVM Loop

Ignoring exceptions, the inner loop of a Java virtual machine interpreter is effectively do {
  fetch an opcode;
  if (operands) 
    fetch operands;
  execute the action for the opcode;
} while (there is more to do);

Opcodes and operands

- The number and size of the operands are determined by the opcode
- If an operand is more than one byte in size, then it is stored in big-endian order
- The bytecode instruction stream is only single-byte aligned
- Not assuming data alignment means that immediate data larger than a byte must be constructed from bytes at run time on many machines

JBC Data Types

- Correspond closely to Java types; conspicuous for its absence is boolean, which in JBC is stored as an int
- This is because it is no more (and is likely to be less) efficient in most real architectures to access a single bit as opposed to a single (32-bit) word – so boolean values are stored as 1 or 0
- Other sub-word storage types (byte, short and char) are promoted to word type for arithmetic operations (implicit promotion, to us) – but that’s in the stack, not in memory
- Operations on these types are, effectively, int operations
### JBC data types

<table>
<thead>
<tr>
<th>Data type</th>
<th>JBC Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>i</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>float</td>
<td>f</td>
<td>32-bit IEEE 754 floating point number</td>
</tr>
<tr>
<td>long</td>
<td>l</td>
<td>64-bit integer – takes 2 stack frames</td>
</tr>
<tr>
<td>double</td>
<td>d</td>
<td>64-bit IEEE 754 floating point number (2 stack frames)</td>
</tr>
<tr>
<td>byte</td>
<td>b</td>
<td>8-bit signed integer</td>
</tr>
<tr>
<td>short</td>
<td>s</td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td>char</td>
<td>c</td>
<td>16-bit unsigned integer or Unicode (UTF-16) character</td>
</tr>
<tr>
<td>address</td>
<td>a</td>
<td>Objects</td>
</tr>
</tbody>
</table>

### JVM stack frames

- Recall that each thread or program has its own JVM stack to store frames
- Frames are created when methods are invoked
- Frame consists of:
  - operand stack
  - local variable table (array)
  - pointer to the runtime constant pool of the current method's class

### JVM stack frames

- Size of both operand stack and local variable table are determined at compile time
- Operand stack stores:
  - operands for opcode instructions
  - operation results
  - return values from methods

### JVM instructions

- Data typing in Java requires type-specific instructions; thus for example, the add instruction comes in four different flavors:
  - iadd: adds integers
  - ladd: adds longs
  - fadd: adds floats
  - dadd: adds doubles
- Similar instructions exist for other arithmetic operations

### Arithmetic instructions

- Each arithmetic instruction works as follows:
  - top 2 elements are popped off stack
  - result is computed
  - result is pushed to stack
- "Elements" may be one or two words in size:
  - ints and floats: 32 bits, single word
  - doubles and longs: 64 bits, two words each

### Arithmetic instructions

- The modulus operation exists only for the integer and long types; the instruction is irem or lrem
- On the high level, modular division is allowed on float and double types, but the result is always a whole number – evidence of implicit type conversion
### Data typing and arithmetic instructions
- Mixed-type expressions must have all operands converted to single data type for evaluation
- Unary conversion operations exist to facilitate this:
  - i2f: converts int to float
  - b2i: converts byte to int
  - etc.
- Always possible to convert between the 4 basic types (i, f, l and d), and anything can be converted to int

### Logical & shift operations
- Operate on integer types only
- Logical operations include and, or and xor; examples:
  - land: and on 2 longs
  - ixor: xor on 2 ints
- Shift operations on ints include ishl, ishr and iushr (unsigned shift right); similar operations exist for longs

### Data access operations
- Load/store instructions exist for the 4 primitive types; transfer values between local variable table & operand stack
  - iload: push int variable on stack
  - dstore: store stack value in local double variable
  - const versions exist for literal value load/store
- Can also load/store objects using aaload/astore

### Object creation & manipulation
- Although both class instances and arrays are objects, the Java virtual machine creates and manipulates class instances and arrays using distinct sets of instructions:
  - Create a new class instance: new.
  - Create a new array: newarray, anewarray, multionewarray.

### Object creation & manipulation
- Access static fields and instance variables: getfield, putfield, getstatic, putstatic
- Load an array component onto the operand stack: iload, laload, faload, daload, aaload, etc.
- Store a value from the operand stack as an array component: iastore, etc.
- Get the length of array: arraylength

### Stack manipulation & method handling
- Several instructions operate directly on the operand stack, including pop, pop2, swap, and several others
- Method invocations are handled by instructions specific to the type of method:
  - invokevirtual: starts an instance method
  - invokestatic: starts a static method
  - invokeinterface: starts a method specified by an interface
- Various return instructions are used to return values from methods (ireturn, dreturn, etc. – also just return for void methods)
Java class files

- Consists of stream of bytes
- Class file data types describe the various fields in the class file format:
  - u1: unsigned 1-byte number
  - u2: unsigned 2-byte
  - u4: unsigned 4-byte
- The next several slides describe the class file format in depth

ClassFile structure

```java
ClassFile
{
  u4 magic;
  u2 minor_version;
  u2 major_version;
  u2 constant_pool_count;
  cp_info constant_pool(constant_pool_count-1);
  u2 access_flags;
  u2 this_class;
  u2 super_class;
  u2 interfaces_count;
  u2 interfaces[interfaces_count];
  u2 fields_count;
  field_info fields[fields_count];
  u2 methods_count;
  method_info methods[methods_count];
  u2 attributes_count;
  attribute_info attributes[attributes_count];
}
```

Fields in Classfile structure

- magic_number: used to identify this file as a class; the magic number is the hex value CAFEBABE (no, I’m not kidding)
- minor_version and major_version are class file versions; values must fall within a range of numbers (defined by Sun) in order to be runnable on a particular JVM
- constant_pool [] and constant_pool_count:
  - constant_pool is an array of string literals, class, interface and field descriptors that are referenced in the Classfile structure
  - constant_pool_count is the size of the constant_pool
- access_flags: set of flags indicating access information (public, private, etc.) about the class or interface

Fields in Classfile structure

- this_class: must be valid index to constant_pool; entry at that index is structure describing (very briefly) the current class
- super_class: 0 if this class is not derived; otherwise, must be valid to constant_pool; entry at index describes the superclass

Fields in Classfile structure

- interfaces[] and interfaces_count: the latter is the number of superinterfaces of the current class; the former is an array whose entries are valid indexes to the constant_pool, where the entries are structures describing all of this class’s superinterfaces
- fields[], fields_count, methods[], methods_count, attributes[] and attributes_count: more of the same
Example descriptor field

- A method descriptor has the following format:
  
  ```java
  method_info {
    u2 access_flags;
    u2 name_index;
    u2 descriptor_index;
    u2 attributes_count;
    attribute_info attributes[attributes_count];
  }
  ```

Digging in a little deeper ...

- The method_info structure (which is itself an entry in the constant_pool) contains another structure, attribute_info[]
- As the name suggests, this structure is an array of method attributes
- Attributes include constant values, code, exceptions and several others; we will confine our discussion to the first two

ConstantValue attribute

- fixed-length structure representing value of a static constant; descriptor is:
  
  ```java
  ConstantValue_attribute {
    u2 attribute_name_index;
    u4 attribute_length;
    u2 constant_value_index;
  }
  ```
- Both indexes refer to the constant_pool, where they must match legitimate entries; value of attribute_length for ConstantValue is 2

Code_attribute

- Contains actual JVM instructions for a method
- Descriptor:
  
  ```java
  Code_attribute {
    u2 attribute_name_index;
    u4 code_attribute_length;
    u2 max_stack;
    u2 max_locals;
    u4 code_length;
    u1 code[code_length];
    u2 exception_table_length;
    (u2 start_pc;
     u2 end_pc;
     u2 handler_pc;
     u2 catch_type;
    ) exception_table[exception_table_length];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
  }
  ```

Code_attribute fields

- max_stack and max_locals give the size of the operand stack and local variable table for the method's frame
- code_length and code[] are the number of instructions and an array containing the instructions themselves

Code_attribute fields

- exception_table[] is an ordered array of exception handler descriptors; exception_table_length is the array size
- start_pc and end_pc indicate the indexes of the code[] array that define the range within which exception listeners are active
Even imagination has its limits

- The JVM, because it isn’t real, has access to theoretically unlimited resources
- Of course, it still has to exist in the real world, and both hardware and the JVM spec itself impose constraints
- The next several slides describe some of these

Size matters

- Consider the following ClassFile attributes:
  - u2 constant_pool_count;
  - u2 fields_count;
  - u2 methods_count;
  - u2 attributes_count;
- All of these are type u2: unsigned 2-byte integer; the maximum size of a 2-byte integer is 65,535
- Thus, each of the arrays described by the various count attributes is limited to ~64K entries

Size matters (again)

- The Code_attribute descriptor includes these fields:
  - u2 max_stack;
  - u2 max_locals;
  - u4 code_length;
- Thus the operand stack and local variable table for a method are subject to the same 64K limit
- The code_length attribute implies that the number of instructions could exceed the limit, but the actual code length is limited by the size of the exception_table

Last word on sizes

- 255 is the maximum number of:
  - array dimensions
  - parameters to a single method
- 65,535 is the maximum length of:
  - identifiers
  - String literals