Java Bytecode – notes & exercises for Lectures 12-13

Recall from Lectures 7-8 (Overview of Java Virtual Machine) the two different ways in which high level programs are processed -- interpreting v. compiling. On these you can also red in [1] – Sec. 5.1, 5.2; and [2] Chap 4.

Also recall combining the two using virtual machine as in Java – [2] Chap 4. Try to understand the cleverness of how it allows a single compiled .class file to run on a wide range of platforms.

The Java Runtime Environment (JRE) is what you call up when you use the "java" command. It opens up a Java virtual machine (JVM), loads some classes, and runs their bytecode on the JVM.

In Lectures 12-13 we saw more details about bytecode.
In Lecture 12:

- Frames: the basic structure within which bytecode is executed, with an operand stack and space for local variables.
- Make sure you understand how every non-static method stores a reference to "this" in its local variable at index 0. (For a static method, index 0 is nothing special - it is just the first available index.)
- Every time a method is called, a new frame is constructed for it: so one frame is current and the rest are dormant (asleep). You can see this working in the factorial example.
- The first letter in a mnemonic often indicates the type (i, l, f, d, b, s, a).
- Local variables are accessed just using their index. The name is usually not needed, though it is kept for the debugger.

In Lecture 13:

- Methods and fields are accessed at runtime via their names and classes: these are called symbolic references, because they use the symbols of the name. It has to be like this, because they might be in different classes, compiled independently. The symbolic references are not put into the bytecode itself, because that would take up too much space in the bytecode. Instead, they are recorded as an item in the runtime constant pool for the class, and the bytecode has an index number for that item. In practice you don't need to know the details of how this works. The disassembled bytecode shows the actual names from the constant pool.
- The example shows all these features coming together. Make sure you understand (i) how the bytecode operates within a frame, accessing the local variables and using the operand stack for calculations, and also (ii) how the different frames interact for the method calls.

References

[1] Goldschlager and Lister: *Computer Science: a Modern Introduction* (Prentice Hall, 2nd edition 1988)

[2] Reynolds and Tymann: *Principles of Computer Science* (Schaum's Outlines, McGraw-Hill 2008)

Using javap

javap takes a compiled (.class) file and produces a readable version, using mnemonics instead of bytecode. This is how to use it from the Windows command prompt, but it will be similar in Linux.

- It's easiest if you first go to the folder where the .class file is.
- Let's say the .class file is `MyClass.class`.
- `javap MyClass` will display a summary.
- To display more you need "flags":
    - -c gets the disassembled bytecode for each method
    - -l gets the slots used for local variables
- Instead of displaying the output, you can redirect it to a file using ">".
- e.g. `javap -c -l MyClass > MyClassDisass.txt`

Exercises

You can try disassembling some simple methods of your own to see how they work. Make sure you try out both some static and some non-static methods too to understand the difference. Here is an example:

Take a look at the file Last.java. It has two versions of a method to find the last element of an array (assuming the array is not empty, i.e. length 0). One is static and uses a parameter, the other is non-static and uses an instance variable (field).

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package introcs;
/**
 *
 * @author Steve Vickers
 */
public class Last {
    private int[] arr;
    public static int lastS(int[] arr){
        return arr[arr.length-1];
    }
    public int lastNS(){
        return arr[arr.length-1];
    }
}
```

For each of the methods `lastS` and `lastNS`,

- Use javap to produce the disassembled output.
- Which local variable slots are used to store the local variable(s)?
- Which lines of bytecode correspond to which parts of Java code?
- Show the entire contents of the operand stack for each line of the bytecode, after the corresponding instruction gets executed.

Solutions

1. First, for the static `lastS`:

Disassembled output:

```
  public static int lastS(int[]);
    Code:
       0: aload_0
       1: aload_0
       2: arraylength
       3: iconst_1
       4: isub
       5: iaload
       6: ireturn
```

The local variables are as follows:

slot	*variable*	*comments*
0	arr	the *parameter,* not the field

The lines of bytecode correspond to parts of Java code as follows.
0,5: `arr[...]`
1,2,3,4: `arr.length-1`
6: `return ...;`

The contents of the stack are as follows. Note that "arr" on the operand stack is a *reference* to the array as an object, not the entire contents of the array. Also, in this static method it is the parameter arr, not the field.

line number	*operand stack*
0	arr
1	arr, arr
2	arr, arr.length
3	arr, arr.length, 1
4	arr, arr.length-1
5	arr[arr.length-1]
6	

2. Now, for the non-static `lastNS`:

Disassembled output:

```
  public int lastNS();
    Code:
       0: aload_0
       1: getfield      #2                  // Field arr:[I
       4: aload_0
       5: getfield      #2                  // Field arr:[I
       8: arraylength
       9: iconst_1
      10: isub
      11: iaload
      12: ireturn
```

The local variables are as follows:

slot	*variable*	*comments*
0	this	Always for a non-static method, slot 0 is this.

The lines of bytecode correspond to parts of Java code as follows.
0,1,11: `arr[...]`
4,5,8,9,10: `arr.length-1`
12: `return ...;`

The contents of the stack are as follows. Again, "arr" on the operand stack is a *reference* to the array as an object. However, this time it is the field (instance variable) `this.arr`.

line number	*operand stack*
0	this
1	arr
4	arr, this
5	arr, arr
8	arr, arr.length
9	arr, arr.length, 1
10	arr, arr.length-1
11	arr[arr.length-1]
12	