[PDF version for printing]

Problem Set 6
Nibbling at Byte Code

Out: 31 March
Due: Monday (beginning of class), 10 April

Collaboration Policy - Read Carefully (similar to PS5)

For this assignment, you may work on your own or with any one other person of your choice **except for anyone you worked with on PS5**. If you work with a partner, you should turn in one assignment with both of your names on it. **If you would prefer to be assigned a partner, send email to *evans@cs.virginia.edu*** (include any constraints or preferences you have on your assigned partner). If a suitable match requests a partner, you will receive a partner assignment. Partners will be assigned using a greedy algorithm based on when requests arrive, so you are more likely to receive a suitable partner assignment if you send in your request early.

You may consult any outside resources including books, papers, web sites and people you wish. You are also encouraged to discuss these problems with students in the class. You must acknowledge any people and outside resources you work with on your assignment. If you discuss the assignment with people other than your partner, you may not take any written materials out of your discussion. It is fine to bounce ideas off other people, but the answers you turn in must be your own.

You are **strongly encouraged** to take advantage of the staffed lab hours posted on the CS216 web site.

> **Submission:** For this assignment, you should submit all your answers (including code) on paper in class on April 10. In addition, if you want to be eligible for the Byte Code Wizard awards you should submit your modified `Mystery.class` files for questions 8 and 10 electronically using the form at *http://www.cs.virginia.edu/cs216/ps/ps6/submit.html* .

Purpose

- Explore the Java Virtual Machine Language
- Learn how Java source files correspond to class files
- Understand how the Java VM ensures code safety

> **Download:** ps6.zip. This zip file contains `jasmin.jar` and `D-Java.exe`, the source files for the election implementation (`BallotDefinition.java`, `CompleteElection.java`, `Election.java`, `ElectionResults.java`, `NoWinnerException.java`, and `Office.java`), and the class files for the optimization questions (`Mystery.class` and `Tester.class`).

Background

This assignment will involve using several new tools:

- Jasmin — an assembler for JVML
- D-Java — a Java disassmbler

We will also use the Java compiler, virtual machine, and class file viewer included in the JDK:

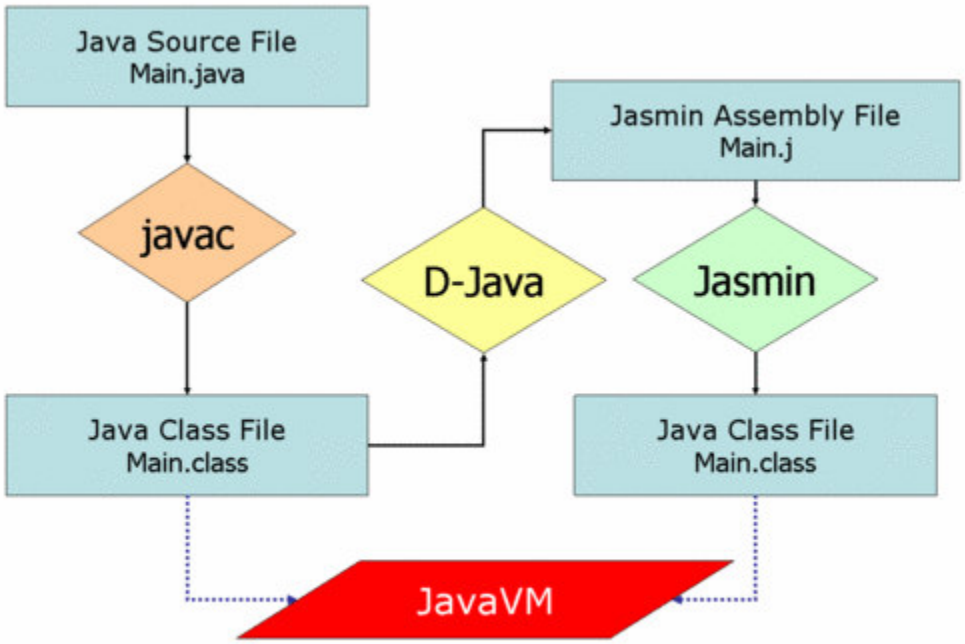
- `javac` — Java programming langauge to JVML compiler
- `java` — the Java virtual machine
- `javap` — Java class file disassembler

Most of you are already familiar with Java from CS101 and CS201. If you do not already have experience with Java, it is strongly recommended that you work with a partner who has some Java programming experience on this assignment.

A comprehensive reference to JVML is *The Java Virtual Machine Specification* by Tim Lindholm and Frank Yellin. A specification for the Java programming language is *The Java Language Specification* (third edition) by James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. (If you remember Java programming from CS201 and CS101, you shouldn't need this.)

Assembling and Disassembling

Sun did not specify an assembly language for the Java virtual machine. An assembly language is a small translation step up from the raw machine language. An assembler converts a (barely) human-readable assembly language program into a binary object file. In this case, we will use Jasmin to convert files in Jasmin's assembly language into Java class files. A disassmbler converts binary files into human-readable assembly language files. We will use D-Java to translate Java class files into Jasmin assembly. We could create Jasmin assembly files from scratch, but it will usually be easier to start by creating a Java source program and using the Java compiler (`javac`) to create a class file, and then using D-Java to disassemble that class file to produce the Jasmin assembly file. The output produced by `javap` is slightly different from D-Java, and is not assembly code that can be assembled by Jasmin. We will use `javap` to examine class files, since it provides more detailed information that D-Java, but use D-Java to disassemble class files into editable Jasmin assembly files. The image below shows the translation paths:



To convert a Java class to jasmin assembly run:

```
] D-Java -o jasmin Name.class > Name.j
```

This produces the output file *Name*`.j`.

To assemble a jasmin file into a class file run:

```
] java -jar jasmin.jar Name.j
```

Prospecting Byte Codes

To gain familiarity with JVML, the first four questions ask you to generate Java programming languages that contain particular JVML instructions or sequences of instructions. Your goal for these questions is to create the simplest Java source code file that the `javac` compiler compiles into a class file containing the given JVML sequence. You answers should be the Java source code you created and a snippet of the generated class file showing the relevant JVML excerpt

(with some context). Use `javap -c class` to view the created class file.

1. Create a Java source code file that compiles to a class file including the `iload` instruction.

2. Create a Java source code file that compiles to a class file containing two consecutive `aload_0` instructions.

3. Create a Java source code file that compiles to a class file containing three consecutive `invokevirtual` instructions.

4. Create a Java source code file that compiles to a class file containing the `wide iload` instruction (`javap` will print it as `iload_w`.

Violating Type Safety

With the Java programming language and the standard Java compiler, all programs are guaranteed to be type safe and memory safe (as long as there are no compiler bugs). This means we cannot treat Objects and integers, or integers as Objects, and can only manipulate data using appropriate methods and operators.

By producing byte codes directly, however, we can violate type safety.

The next three questions assume that you were hired by LiveMeek, an unsuspecting vendor to produce a fancy animation that runs at the end of the election to impress the election officials who will decide which voting machine to purchase. Unbeknownst to the vendor (or the election officials), you are an associate of Mooch, a stray underdog candidate in the upcoming Dog Catcher election. To ensure the election of Mooch to the Dog Catcher position, and guarantee the safety of all Mooch's stray dog friends, your task is to violate type safety to help Mooch steal the election.

Before printing out the election results, the election code will call `CompleteElection.displayAnimation()`. Mooch has asked you to create an implemention of `CompleteElection.displayAnimation` that will set Sarge's vote total to 0 to ensure Mooch's victory (or at least a tie). (Of course, we know no UVa student would ever do anything so heinous as fix an election, but this is just an exercise.)

LiveMeek assumes it is safe to hire you to implement `CompleteElection`, since the sensitive `ElectionResults` object will not be visible inside the `CompleteElection` class. You know, however, that they will turn off the bytecode verifier when they run the election. The ps6.zip file contains the implementation of the LiveMeek voting machine. The files `BallotDefinition.java`, `Election.java`, `ElectionResults.java`, `NoWinnerException.java`, and `Office.java` are part of their implementation and you cannot change them. Your goal is to create a `CompleteElection.class` file that will enable Mooch to steal the election and ensure freedom for stray dogs everywhere.

5. To tamper with the election, you will need to find out where the `ElectionResults` object `e` is stored in memory. Where is `e` stored? (Explain how you found out, and include the code you used.)

Hint: You may want to start by modifying the `Election.java` to do:

```
int i = 3;
System.out.println(i);
```

before the call to `CompleteElection.displayAnimation()`. Then use `D-Java` to generate the corresponding Jasmin assembly code, and figure out how to modify it to find out the location of `e`.

6. Modify `CompleteElection.j` to obtain a reference to the election object.

7. Finish your implementation of `displayAnimation` to change Sarge's vote total to 0. If you are sucessful, running

```
java -noverify Election
```

(on the original Election class) will produce:

```
The dog catcher is: Mooch
```

Your code may assume that the "Dog Catcher" office is always the first office in the election, and "Sarge" always the first candidate for the office.

Your solution should not require making any changes to any class besides `CompleteElection`. To develop your solution, you will probably want to make changes to other classes, though, so you can use to Java compiler to generate byte codes similar to those you will need to use in your solution.

If you can change the election results only by changing `CompleteElection.j` without requiring the `-noverify` option (or doing any physical damage to the ITC lab machines!), that is worth a Double Gold Star bonus.

Optimizing Classes

For the rest of this assignment, your goal is to optimize a Java class. The challenge is you are not given a specification for what the class implements or its source code.

The ps6.zip file contains two Java classes: `Mystery.class` and `Tester.class`.

Your goal is to make Mystery.class smaller and faster. You can make any changes you want as long as your modified Mystery class still passes the test cases executed by `java Tester`.

The provided Tester class includes randomness so does not run exactly the same tests every execution. If you want to control the tester to make reproducible tests for your debugging, you can run it using `java Tester [seed] [number]` where seed is the seed used by the pseudorandom number generator (it doesn't matter what value you use for this, just pick a number) and [number] is the number of tests to run. If no parameters are given, Tester will run 1000 tests.

To modify the class file, you should use D-Java to disassemble it first, and then use jasmin to assemble a new class file. We recommend keeping a careful record (or backup copies) of the changes you make and re-running the tester regularly, so you notice right away if a change alters the expected behavior and can revert to a known good version.

8. Modify `Mystery.class` to minimize the size of the class file. Your modified code should run in the standard Java virtual machine (which includes passing the bytecode verifier). Your answer should explain the changes you made.

9. Modify `Mystery.class` to minimize the running time of calls to `Mystery.doIt`. Your modified code should run in the standard Java virtual machine (which includes passing the bytecode verifier). Your answer should explain the changes you made.

10. Modify `Mystery.class` to minimize the class size and running time of calls to `Mystery.doIt`. Unlike questions 8 and 9, for question 10 the resulting code will execute without bytecode verification (e.g., using `java -noverify Tester`). Your answer should explain clearly the changes you made.

Remember to submit your Mystery.class files for question 8 and 10 using the form at *http://www.cs.virginia.edu/cs216/ps/ps6/submit.html* to be eligible for the Byte Code Wizard awards.