We've talked in class about the profile-guided optimization mechanism that allows a Java VM to focus its compilation efforts only on those parts of the code that are likely to give the best returns. In this assignment, you will implement the decision engine in the SimpleJava VM that determines what code should be optimized, and the Class Hierarchy Analysis (CHA) that tells us what optimizations are allowed.

For the purposes of this assignment, you will not be implementing the compiler phases that actually optimize the code.

The SimpleJava JIT compiler transforms code at the bytecode level to perform the inline optimization that we discussed in class. It is not a compiler in the strictest sense since it produces bytecode rather than machine code, but working at the bytecode level allows you to see the inline operation and how it affects the code being interpreted.

Getting Started

You will set up and run the SimpleJava VM just as you did in Assignment 1. As before, the zip file that contains these instructions also has three directories: `simplejava-assignment2`, `class-lib` and `sample-code`. Follow the instructions posted on Canvas to build all three projects.

The code distributed in the zip file contains implementations of the seven bytecodes that made up the previous assignment, so you should be able to run all of the sample code that was distributed with the Assignment 1. The sample code directory distributed with Assignment 2 contains a few additional test cases that exercise corner cases relevant to JIT compilation. Remember that the sample code is a starting point, and that grading will focus on your complete implementation rather than just on whether you pass the sample cases.

Changes to the SimpleJava VM

There have been a few additions to the SimpleJava code base for this assignment (as well as fixes for the handful of bugs that people came across while working on Assignment 1). There are some new interfaces in the execution package (`edu.harvard.cscie98.simplejava.execution`) that define the JIT profiler and compiler. The Compiler is specified in `JitCompiler.java`, and the profiler in `DynamicProfiler.java`. There is also an interface `InlineOptimization` that defines an inline operation. You should familiarize yourself with these interfaces, and understand how they interact.

There is a new package in the implementation module: `edu.harvard.cscie98.simplejava.impl.jit`. This contains implementations of the three new interfaces. `BcelInlineOptimization.java` contains the code to inline one method inside another, `JitCompilerImpl` implements the main body of the compiler, and `DynamicProfilerImpl` contains a skeleton implementation of the profiler.

In addition to the new code, there is also an additional command line argument: `-jitThreshold`. This value is passed to the profiler to determine how many times a method should be called before it is optimized.

Assignment

Your job in this assignment is to implement the `DynamicProfiler` interface. Apart from the requirement that your implementation must conform to the contract specified in the interface, you are free to define whatever data structures and algorithms you see fit. You should not need to modify any code outside of the `DynamicProfilerImpl.java` file, although you are free to create new files containing helper classes if you wish.

The seven methods that you will implement inside `DynamicProfilerImpl` will be called by the `JitCompilerImpl` class in order to register interpreter and class loader events, and to make compilation decisions.

You are free to implement the methods of `DynamicProfilerImpl` in any order that you choose, but a good approach could be to divide the assignment into three parts that you can reason about separately:

1. **Compilation Threshold**. This first step involves implementing `methodCalled`, `setCompilationThreshold` and `shouldCompileMethod`. Together they implement the decision process that determines whether a method should be compiled.
2. **Class Hierarchy Analysis and Inlining**. For the next step you will analyze the class hierarchy whenever a class is loaded into the VM, and determine what methods on those classes can be inlined. You will implement `analyzeClassHierarchy` and `canInline` at this point.
3. **Deoptimization**. Once you've implemented the other two parts, you may notice that you can construct a test case in which you trigger compilation and then class loading in a sequence that leads to incorrect program behavior. You will solve this problem by flagging methods with optimizations that are invalidated by subsequent class loading events. The JIT compiler will then deoptimize and recompile those methods in order to maintain a correct application. You'll implement this analysis in `getMethodsWithInvalidOptimizations`.

Tips and Suggestions

Remember that compilers shouldn't change program semantics (i.e. the behavior of the application should be identical before and after the optimization pass). Take a look at the output of the sample code running under the SimpleJava VM before you make any changes, and make sure that you don't change its behavior through the JIT.

Vary the compile threshold command line argument to simplify your testing. If you set it to a low value you can trigger compilation more quickly which may give you a shorter program run to track down any bugs.

Pay attention to the Javadoc comments on the `DynamicProfiler` interface. They describe the detailed behavior expected from your implementation. The project will be graded based on the semantics of the interface.

In a production VM it is likely that the Class Hierarchy Analysis will be done incrementally to minimize the work that has to happen when a class is loaded. For this assignment it is perfectly acceptable for the analysis to be performed from scratch whenever a class is loaded. This is less performant, but it simplifies the implementation a lot.

When implementing the CHA, check the interface to `VmClassLoader` for a way to get all of the classes loaded in the system. Note also that the contract for `VmMethod` guarantees describes the semantics for the `hashCode` and `equals` operation between VmMethods. This allows `VmMethod` instances to act as keys in `HashMaps` or to be stored in `HashSets`.

Submitting the Assignment

As in Assignment 1, the submission for this assignment will be a single patch file. The patch will be generated by Git, and will show the changes that you've made against the original `simplejava-assignment1` repository distributed with this file. There is no need to turn in the `sample-code` or `class-lib` directories.

There are instructions on Canvas describing how to properly format a Git patch for submission. Follow the submission instructions carefully; initial grading of your submission will be by script, and a failure of that script due to a badly formatted patch file will cost you points. If you sanity-check your patch using the instructions on Canvas you can be confident that it is correctly formatted.

There will also be an optional survey posted on Canvas asking roughly how long you spent on the assignment, and for any comments. Completing the survey won't affect your grade (I'm only interested in the aggregate and anonymized data), but it will help to calibrate the difficulty of future assignments.

Feel free to post any questions or issues around the patching and submission process on the forum.