We've talked in class about the abstract computer that Java bytecode targets. In this assignment you will gain some hands-on experience by implementing parts of the SimpleJava interpreter. The SimpleJava VM codebase distributed with this assignment contains a partial implementation of the interpreter. Your task is to correctly fill in the missing bytecode implementations to complete the interpreter.

Getting Started

The zip file that contains these instructions also has three directories: `simplejava-assignment1`, `class-lib` and `sample-code`. Follow the instructions posted on Canvas to build all three projects. Once you have successfully built the code, use the SimpleJava VM to run the `Empty` class in the sample code (remove line breaks):

```
mvn -q exec:java -Dexec.mainClass="edu.harvard.cscie98.simplejava.SimpleJavaVm"
 -Dexec.args="<path>/class-lib/target/classes:<path>/sample-code/target/classes
 edu.harvard.cscie98.sample_code.Empty"
```

When the command runs successfully (you'll know because the VM will print `Run complete. Printing output buffer`, although there's no output to print) you'll see that nothing particularly interesting has happened; this is to be expected. Check the Empty.java source and you'll see that it just contains an empty main method. That's not to say that nothing happened when you ran the code. You can see what's going on inside the interpreter by printing verbose interpretation:

```
mvn -q exec:java -Dexec.mainClass="edu.harvard.cscie98.simplejava.SimpleJavaVm"
 -Dexec.args="<path>/class-lib/target/classes:<path>/sample-code/target/classes
 edu.harvard.cscie98.sample_code.Empty -verboseInterpreter"
```

You should now see the output of the interpreter:

```
[interpreter]  PC: 0, Stack:  [] Locals: [0x100000 ] return
[interpreter]  Method returning void
[interpreter]  Empty stack. Exiting
Run complete. Printing output buffer
====================================
[interpreter]  PC: 0, Stack:  [] Locals: [] sipush
[interpreter]  PC: 3, Stack:  [1024] Locals: [] anewarray
[interpreter]  PC: 6, Stack:  [0x1005c8] Locals: [] putstatic
[interpreter]  PC: 9, Stack:  [] Locals: [] return
[interpreter]  Method returning void
[interpreter]  Empty stack. Exiting
Standard Output
===============
```

There are two methods running here. The first is the body of the `Empty.java` main method. It has exactly one bytecode, `return`, that exits the method. At that point the application is complete. The remaining four bytecodes that are executed after the application is complete are a part of the VM's termination process. In order to print any messages that were sent to `System.out`, the VM inspects the static state of the `java.io.PrintStream` class. Since the `Empty.java` class did not refer to `System.out`, the class was not loaded during the run of the application. The code that executes after the run finishes is the static initializer for the `PrintStream` class. You

can see how this mechanism works by looking at the `SimpleJavaVm` class, as well as `PrintStream` in the class library.

Once you've got the `Empty` class up and running, you're in a good state to start working on the assignment. If you come across any issues building and running the VM, post a message to the discussion forum under "Working with the SimpleJava VM". Feel free to help out your classmates on the same forum page to boost your participation grade.

The Interpreter

Familiarize yourself with the SimpleJava VM's code structure, referring to the documentation on Canvas as well as the Javadoc site generated from the code (see Assignment Zero for instructions on generating the documentation). You can find the code for the SimpleJava interpreter under `edu.harvard.cscie98.simplejava.impl.interpreter`. The main interpreter loop is implemented in `SwitchInterpreter.java`, and the logic for each bytecode instruction is in the `bytecodes` package. You will see that some of the bytecode classes are missing implementations; they are marked with a `TODO` comment, and will throw a `RuntimeException` if called. The following table indicates the bytecodes to be implemented, along with sample code that depends on them. The ordering of the bytecodes is recommended; you are free to implement the bytecodes in whatever order you wish, but following the order in the table will give you a smoother learning curve and let more sample code run sooner.

Bytecode	**Applications**
IADD	Addition, StaticFieldAccess
IINC	
GOTO	Loop, Multiplication, Arrays
GETFIELD	
PUTFIELD	FieldAccess
INVOKESTATIC	Fibonacci, Quicksort
INVOKEVIRTUAL	HelloWorld

Once you have implemented all of the bytecodes you should be able to run any of the sample code applications. Be sure to delete the `RuntimeException` when you have implemented the bytecode.

Expected Output

To make sure that you're on the right track, here's the partial expected output of the `Addition.java` sample program after implementing the `IADD` bytecode and running using the `-verboseInterpreter` option:

```
[interpreter]  PC: 0, Stack:  [] Locals: [] bipush
[interpreter]  PC: 2, Stack:  [42] Locals: [] putstatic
[interpreter]  PC: 5, Stack:  [] Locals: [] return
[interpreter]  Method returning void
[interpreter]  Empty stack. Exiting
[interpreter]  PC: 0, Stack:  [] Locals: [0x100000 ] getstatic
[interpreter]  PC: 3, Stack:  [42] Locals: [0x100000 ] bipush
[interpreter]  PC: 5, Stack:  [42, 10] Locals: [0x100000 ] iadd
[interpreter]  PC: 6, Stack:  [52] Locals: [0x100000 ] putstatic
[interpreter]  PC: 9, Stack:  [] Locals: [0x100000 ] return
[interpreter]  Method returning void
[interpreter]  Empty stack. Exiting
Run complete. Printing output buffer
====================================
```

```
[interpreter]  PC: 0, Stack:  [] Locals: [] sipush
[interpreter]  PC: 3, Stack:  [1024] Locals: [] anewarray
[interpreter]  PC: 6, Stack:  [0x100700] Locals: [] putstatic
[interpreter]  PC: 9, Stack:  [] Locals: [] return
[interpreter]  Method returning void
[interpreter]  Empty stack. Exiting
```

By looking at the source code for the `Addition` program and by running `javap` on its compiled class, you should be able to follow the steps that the interpreter takes. The first set of three bytecodes executed are the static initializer for the `Addition` class (initializing the `intVar` static field). The second set of bytecode show the execution of the main method, and the third is the static initializer for `PrintStream`, as above.

Submitting the Assignment

For this and all remaining assignments, the submission 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.