In this assignment you will add exception handling to the SimpleJava VM. You will implement the `ATHROW` bytecode that allows applications to throw custom exceptions, and add runtime exceptions for error conditions in the interpreter.

For this assignment it is particularly important to remember the distinction between the host Hotspot VM and the guest SimpleJava VM. Your job is to implement exception management in the SimpleJava interpreter, which will involve analyzing the stack and running methods. If you find yourself typing `throw new` as part of the SimpleJava VM code in your solution, you may be approaching the problem incorrectly.

Getting Started

As before, the zip file that contains these instructions also has three directories: `simplejava-assignment4`, `class-lib` and `sample-code`. The SimpleJava VM contains only a minor code refresh from the version that was distributed with Assignment 3. The VM is back to using the `InfiniteMemoryManager` implementation for this assignment in order to avoid GC activity when you allocate memory.

JIT

The JIT compiler is turned off for this assignment (as in Assignment 3, the DynamicProfilerImpl class contains a basic skeleton that never performs optimization). This is particularly important to understand in this assignment when you start working out stack traces. Part of the requirements for the stack trace is that it includes the original file name and line number from which the exception was thrown. Working this out is far more complicated in the presence of inlining, since you have not only to track the original line numbers that corresponded to the inlined code, but also the files in which they were defined.

For the sake of this assignment, you may assume that there is never any inlining performed.

Exceptions

As we discussed in class, when an exception is thrown the currently-executing method is interrupted. There are two possible outcomes:

1. The exception is caught by the current method. In that case control in the method is transferred to the exception handler code.
2. The exception is not caught by the method. The stack is then unwound, with each method inspected for an appropriate handler block.

If an exception is not caught by any handler the program terminates.

Java also allows us to define a `finally` block of code that runs regardless of whether the `try` block threw an exception. We discussed in class how the `finally` block is implemented.

ATHROW

The first part of the assignment is to implement the `ATHROW` bytecode. The semantics for the bytecode are largely the same as defined in the Java VM spec, with some minor differences:

- You may assume that the verifier guarantees that the object on top of the stack will always be a reference, and that the type verifier has determined that it is a subtype of `Throwable`. You do not need to handle the case when the bytecode attempts to throw an object that is not `Throwable`.
- SimpleJava does not have any synchronized or locking semantics. You may ignore the part of the specification that refers to monitors.

- SimpleJava does not support asynchronous exceptions.

Note that you cannot assume that the object on top of the stack is not `HeapPointer.NULL`. Your code must correctly handle `finally` blocks.

Stack Traces

When an exception is thrown, you must provide a stack trace that indicates where the exception occurred. There are two methods in `InterpreterUtils` that will help with this: `initializeStackTrace` and `addLineToStackTrace`. These methods construct stack trace strings from the information that you provide. The first method must be called once for every exception object, and the second must be called for every method in the stack trace. Note that correct format for stack traces will be part of the grading criteria for this assignment, so it is strongly recommended that you use these methods rather than implementing your own traces.

The stack traces that you generate should contain the same information as you would expect to see when running a piece of code in the Hotspot JVM. As part of your testing, you may want to run the same program under both Hotspot and your own SimpleJava VM implementation to make sure that file names and line numbers are the same.

Runtime Exceptions

Once you have implemented `ATHROW` with stack traces it's time to add the various exceptions that can be thrown by the VM itself. You will add code to throw exceptions to the following bytecodes:

NullPointerException

- Putfield
- Getfield
- ArrayLength
- AALoad
- AAStore
- IALoad
- IAStore
- InvokeSpecial
- InvokeVirtual
- Athrow

ArithmeticException

- IDiv
- IRem

ArrayIndexOutOfBoundsException

- AALoad
- AAStore
- IALoad
- IAStore

AbstractMethodException

- InvokeSpecial
- InvokeStatic
- InvokeVirtual

Remember to factor common code out to `InterpreterUtils`.

Tips and Suggestions

Pay attention to the state of the stack once you throw an exception. The JVM spec is particular about what should be on the stack if an exception is caught inside the same method.

Remember that a null `HeapPointer` is always represented by `HeapPointer.NULL`. This value is a singleton, so you can use == in comparisons. Do not compare `HeapPointers` to `null`.

SimpleJava defines a *primordial method* at the root of the JVM stack, which is there to catch any `Throwable` that has not been caught by a handler block. This way the exception throwing code does not have to handle the case where there is no catch block for a given exception. Your stack trace should not include the primordial method, which can be identifed by the fact that it does not have a defining class or line number table.

Note the JVM specification for exception catch ranges. The start bytecode of a range is inclusive, while the end is exclusive.

When determining whether an exception handler block is active for a given method call, remember that the program counter was incremented by the invoke instruction and so points to the next instruction. If a handler ends at the invoke instruction you may miss it unless you compensate. Note that all invoke instructions are three bytes long.

As always, remember that the sample code is a starting point for your testing. There may be cases that are not covered by the sample code, so you should write your own test programs to cover them.

Submitting the Assignment

As in previous assignments, 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-assignment4` 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.