We've spent a lot of time in class over the past few weeks talking about memory management in the Java VM - now it's your turn to implement one of the algorithms that we've discussed. For this assignment you will implement Cheney's algorithm; a stop-the-world copying semi-space collector that evacuates all live objects from one half of the heap to the other in a single pass. There is a summary of the algorithm below, and remember that the slides and lecture videos of our discussion in class are available on Canvas.

Getting Started

By now, setting up and running SimpleJava should be familiar. As before, the zip file that contains these instructions also has three directories: `simplejava-assignment3`, `class-lib` and `sample-code`. Follow the instructions posted on Canvas to build all three projects.

The code in the zip file contains implementations of the bytecodes from Assignment 1, and a very primitive implementation of the profiler from Assignment 2. The profiler simply returns false for all compilation decisions, meaning that the JIT compiler never runs. Activity by the JIT compiler shouldn't have any effect on your garbage collection algorithm, but eliminating compilation will simplify things when you debug using interpreter traces.

Cheney's Algorithm

The memory management algorithm that you will implement has the following features:

- **Bump-pointer allocation**. Since the garbage collector compacts the heap, there is no danger of fragmentation and so we can use the simple (and efficient) bump pointer allocation technique. Take a look at the existing `InfiniteMemoryManager` to see how to set up a bump pointer allocator in the SimpleJava VM.
- **Semi-space**. The heap is divided into two equally-sized regions. This way in the worst case when all objects survive a collection there is space to copy them all. You can assume that the heap size passed to your code is divisible by two.
- **Stop the world**. The algorithm is neither concurrent nor parallel, so it requires that all application threads are suspended before the garbage collection process begins (and restarted once collection is complete).
- **Single pass**. The GC combines the marking and copying phases of the collector into a single pass, using the two-finger algorithm to eliminate the need for an external gray stack. Refer back to the lecture slides if you are unsure about the two-finger algorithm and how it lets us combine the phases.

Garbage Collection in SimpleJava

You will recall that the first step in tracing the heap for live objects is to identify the root set for the VM. This is implementation-dependent, and is determined by whether the VM stores internal data structures on the heap or off to the side. In SimpleJava, there are three locations for root references:

- **Stack and local variable references**. These are the references that you manipulated when implementing bytecodes such as `PutField` or `InvokeVirtual` in Assignment 1. Stack and local variables are managed by the `JvmThread`.
- **Static references**. These are references stored once per class in static variables. Static variables are managed by the `VmClassLoader`.
- **Interned strings**. Recall that the a VM can internalize certain constant values to save memory and to make comparisons faster. The SimpleJava VM interns `String` constants and stores the result on the heap. You will need to scan the intern table to find all internalized strings. Internalized strings are managed by the `ObjectBuilder`.

Since the GC algorithm for this assignment involves moving objects, you will have to update root references once you have copied the object to which they refer. In a native language such as C, you would implement this using a pointer-to-a-pointer (`void**`). That way you can manipulate the memory location containing the pointer, rather than the pointer itself. In Java, since we don't have direct access to memory, we have to simulate the same operation. Rather than having you digging through stacks, class objects and the intern table to update references, SimpleJava uses the `ReferenceLocation` interface to read and write root `HeapReference` values.

If the garbage collector runs and is unable to free up enough memory to fulfill a memory request, the JVM specification says that it should throw an `OutOfMemoryError`. We don't have the mechanism to throw exceptions to code running inside the SimpleJava interpreter yet (that'll come in Assignment 4), so your code can simply throw a `VmInternalError` that will terminate the VM.

For your implementation you should install forwarding pointers into the object header's class descriptor word. We talked in class about forwarding pointers, and how we can use bit stealing to indicate whether a header word contains a forwarding pointer or a class descriptor. However, since Java doesn't give us direct access to pointers we can't implement the bit stealing trick (we'll talk later in class about how production Java-in-Java VMs get around this). For this assignment, you can take advantage of the fact that the SimpleJava header stores its contents as `Object`, and simply replace the `TypeDescriptor` object with a `HeapPointer`. You can use the `instanceof` operation to distinguish between the two.

Finally, since Java doesn't give us direct access to pointers, the SimpleJava VM simulates a continuous heap by tracking memory addresses in the `Heap` class. When you create objects using the `ObjectBuilder` interface the implementation ensures that the new object is properly registered with the `Heap`. The internals of the `Heap` class should not be relevant to your implementation, but you may want to explore the code to help with debugging and in order to write better unit tests.

Tips and Suggestions

You are likely to spend more time in this assignment debugging than writing code to begin with. This is a basic fact of GC development - bugs tend to be subtle, and often don't show up until long after your garbage collector has finished running. Experienced GC developers try to mitigate this by building self-checks into their code. For example, a very common bug is when your code somehow fails to update a reference on the heap. You can check for this case by doing a linear scan over the new space at the end of the collection to see whether any object contains a pointer to the old space. If so you know that your code has a bug, and you can catch the error earlier than if the program crashed some time later. If you implement verification stages in your code, remember to have a way to turn them off before submitting (for example, you may have a `DEBUG` constant that you check before verifying).

On a related note: running a garbage collection algorithm inside a VM tends to be all-or-nothing. Either the algorithm works, or it crashes the program. Writing unit tests helps you to verify the various parts of the collector in isolation, and lets you set up specific test cases that will exercise your collector in controlled environments. As before, unit tests aren't specifically required for this assignment. However if you invest some time in testing your code along the way you are likely to build a better solution faster, and it'll give some insight into your approach in order to award partial credit.

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-assignment3` 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.