For this final assignment, you will build a generational collector in SimpleJava. It will consist of a fixed-size nursery and a mature space managed by a mark/sweep collector. There are quite a few parts to implement this time around, so I'd suggest that you approach the problem in stages. That way even if you don't manage to get everything running perfectly your individual components can be graded by themselves for partial credit.

The components below are listed in a suggested implementation order.

### Remembered Sets

The write barrier implementation adds one interface (`vm.memory.WriteBarrier`) and one implementation class (`impl.memory.memorymanager.GenerationalWriteBarrier`). An instance of the `GenerationalWriteBarrier` class is created when the VM is initialized, and is passed to the `StackInterpreter`. There are two things that you have to do when implementing the write barrier:

- **Call the barrier on pointer writes**. There are two bytecode instructions that could write references to the heap. You will need to instrument the `interpret` methods for those two bytecodes in order to store writes to the remembered set. Remember that it is only necessary to record pointer writes.
- **Implement the remembered set**. You'll need to fill in the skeleton implementation for the `GenerationalWriteBarrier` class to match the definintion in `WriteBarrier.java`. Remember to record only references that go from the nursery to the mature space.

You can work out which region a pointer is in based on the `HeapParameters` object passed to the constructor, which indicates the heap base pointer, as well as the heap size and the fixed nursery size (both in bytes). Alternatively, you could create methods in the `GenerationalWriteBarrier` class to pass the nursery and mature regions to test for pointer locations (although don't change the `WriteBarrier` interface without discussing it with me first).

### Non Contiguous Region

Before you can implement a mark/sweep garbage collector, you'll need an allocator that can handle fragmentation. If you look in the `vm.memory.Heap` interface you will see a new method: `getNonContiguousRegion`. This works in the same way as `getBumpPointerRegion` from Assignment 3. In this case, however, you will implement the region. There is a skeleton class in `impl.memory.heap.NonContiguousRegion`. You should implement your region to use a first-fit algorithm, keeping track of allocated objects and holes in the address space using the data structure of your choice.

The `Region` interface contains the basic operations that your region must support (allocation and free), and a method to determine whether a pointer refers to an address within the region. You will also need to implement methods to scan through the region in order to implement the sweep phase of the mark/sweep collector. These methods can be implemented on the class rather than the interface, and you are free to choose the signatures.

### Mark Sweep Collector

Once you have implemented the non-contiguous region you will be ready to implement the mark/sweep collector that will manage the mature space. The mark/sweep collector has the following characteristics:

- **Non-contiguous allocation**. You will use the non-contiguous allocator that you implemented in the previous step to allocate and free objects. Unlike the garbage collector in Assignment 3, your mark/sweep collector will only require one region.
- **On-demand GC**. Remember that this collector will be used to manage the mature space of a generational collector; once you have finished the assignment this allocator will only be used during collection of the nursery. Taking the approach from Assignment 3 where GC was triggered when the heap ran out of memory would be

problematic, since by definition the region will only ever run out of memory during a minor GC. This means that as well as the set of roots that you used before, you would also have to account for references inside the nursery (some of which may have been forwarded, meaning that you cannot scan the original object). To simplify implementation, we will instead say that GC is only triggered explicitly by calling the (newly-added) `garbageCollect` method.

- **Stop the world**. The algorithm is neither concurrent nor parallel, so you can assume that the heap will not be modified by any other thread.

If the allocator is unable to complete an allocation request, your code should throw a `VmInternalError` (the Hotspot-level exception, not one thrown through the SimpleJava stack as you did in Assignment 4).

The root set for the mark/sweep collector is the same as in Assignment 3:

- **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**. Constant strings that are stored once per VM instance. Interned strings are managed by the `ObjectBuilder`.

Once you have completed this part of the assignment, you will have a functioning mark/sweep garbage collector. I would recommend testing it in isolation from the generational collector in order to track down any bugs. To test the GC, you may want to add temporary logic that triggers GC when an allocation request fails. Once you've done that, your mark/sweep collector can be dropped in as a replacement for the SemiSpace collector or the new Generational Collector. See the commented-out code in `SimpleJavaVm.java` in order to switch garbage collectors.

### Nursery Collector

You should implement the nursery collector in `impl.memory.memorymanager.GenerationalMemoryManager.java`. Unlike the mark/sweep collector, it will not run stand-alone.

The nursery is managed using a copying collector that promotes all live objects to the mature space during a minor collection. It will be largely similar to the semi-space collector that you implemented in Assignment 3, although with some important exceptions. The list below is not exhaustive, but it covers the major differences between the algorithms:

- The root set for the nursery includes the three sets enumerated above, as well as any references from the mature space. You collected these references when you implemented the remembered set inside the write barrier.
- The nursery consists of a single region, with surviving objects promoted to the mature space. When a minor GC is complete, the nursery is reset for the next allocation.
- A minor collection requires a gray stack. In the semi-space collector we could use the two-finger algorithm to use the to-space as an implicit gray stack. However since the mature space contains objects copied in previous collections, that algorithm will not work. Instead you should use an explicit gray stack that is allocated on the Hotspot heap (you don't need to worry about the space overhead of the gray stack for this assignment).

For this assignment, you will trigger major collections manually (see the section on mark/sweep above). To avoid having to scan the nursery for roots, you can trigger the mature collection immediately after a minor collection is complete. That way all live objects are in the mature space, and the only roots are those listed above.

Since the mature space is managed by an algorithm that allows for fragmentation, it is not sufficient to trigger a major collection when the available space in the mature region is equal to the size of the nursery. The likelihood of all objects fitting into the fragmented mature space is very low, meaning that the last minor collection is likely to fail. Instead, we will use the heuristic of triggering a major collection when the available space in the mature region is less than twice the size of the nursery. There will still be occasions when fragmentation causes this heuristic to fail, but they should be rare and can be compensated for by increasing the heap size.

### Notes

- You should lay out your heap using two regions that are contiguous in the VM's memory space. In other words, your nursery should start at the location indicated by the base pointer in the `HeapParameters` argument, and your mature space should begin immediately afterwards.
- In order to implement the mark/sweep collector, there are two new methods added to `ObjectHeader`: `getMarkBit` and `setMarkBit`. These methods allow you to explicitly set or clear mark bits on the header without worrying about bit stealing.
- Remember that the remembered set is only concerned with references from the mature space to the nursery on the heap. It should not contain references from nursery to mature, from nursery to nursery or from mature to mature. It also does not track references from the stacks or local variables.

### 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-assignment5`, `class-lib` and `sample-code`. Follow the instructions posted on Canvas to build all three projects.

The code in the zip file contains implementations for Assignments 1, 3, and 4, and the same primitive implementation of the profiler from Assignment 2 that we've seen in previous assignments. The profiler simply returns false for all compilation decisions, meaning that the JIT compiler never runs. As before, any activity by the JIT compiler shouldn't have any effect on your garbage collection algorithm, but eliminating inlining will simplify things when you debug using interpreter traces.

### 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-assignment5` 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.