Name: _____

IB Computer Science 2
Memory allocation worksheet

Download `TraceAllocations.zip` to the folder where you usually write activity code, and extract it. To make macOS trust the library, run the following command:
`xattr -d com.apple.quarantine java libTraceAllocations.dylib`

Memory allocation

Whenever a program needs memory, it must **allocate**, or request, at least as many bytes as it needs. This places a reservation on an appropriately-sized chunk of memory, preventing it from being used for any other purpose until the program explicitly **frees**, or deallocates, it.

The `Alloc.java` program prompts for a number of kilobytes of memory to allocate. It then allocates that many one-kilobyte arrays. It uses the `TraceAllocations` library to intercept and display the allocations and frees performed by the JVM. Run it using the command
`./trace Alloc.java`
(note the `./` before `trace`).

1. Try requesting 10000 kilobytes, or approximately 10 megabytes. Looking at the total amount of space that gets allocated, what must the JVM be doing with the allocations when each new array is created?

2. Now request just 10 kilobytes. After rerunning a few times, what is the minimum number of allocations you saw? What does this reveal about where in memory the JVM can place newly-created arrays?

3. Comment out the loop and request a *single* 10000-kilobyte array. After rerunning a few times, what do you notice about the allocation sizes? Where is the JVM placing the array?

4. Change the array size to one gigabyte. What do you notice about the allocation sizes? Where is the JVM placing the array?

Garbage collection

You just caught the JVM reusing existing memory allocations for “new” arrays. In fact, for sufficiently small arrays, it was using memory that it had allocated even before the `TraceAllocations.start()` call that started tracing.

In modern programs, memory allocation is a collaboration between the language runtime environment and the operating system. You may have realized that you have never needed to free memory in your programs; this is because Java, like most modern higher-level languages, uses a memory management technique known as **garbage collection**. As a programmer, whenever you are done using a reference type, you simply let the variable that points to it go out of scope. The JVM periodically runs a maintenance task called the garbage collector, which marks any reference types with no remaining references as available for reuse.

This is why repeatedly executing `byte[] unused = new byte[1024]` was not performing enough allocations to hold all of the created arrays. Because `unused` went out of scope at the end of the loop body, the space that had belonged to the array became theoretically available for reuse. Eventually the garbage collector ran and reclaimed the space. In fact, the memory allocations you saw “from the loop” were actually generated by the garbage collector itself, and not used to store the actual arrays!

You probably also noticed that creating a single 10000-kilobyte array often did not perform any allocations at all. This indicates that there was enough extra space already managed by the garbage collector that the JVM did not need to request any memory from the operating system. It also suggests that the garbage collector did not run, because the program did less work and created fewer reference types.

5. Look back at the output from allocating a one-gigabyte array. Notice that the JVM did not free all of the memory. Try adding a `System.gc()` call before `TraceAllocations.stop()` to request a full garbage collector run. What happens?

6. What do you think happens to the memory that the JVM never frees, and when?

If you have extra time…

Test your prediction. Add another `readLine()` at the end of the program to prevent it from immediately exiting. Open the macOS Activity Monitor application and watch its App Memory number as you use the updated program.