CS61B: Lecture 39
Wednesday, April 25, 2012

Today's reading: Goodrich & Tamassia, Sections 14.1.2-14.1.3.

GARBAGE COLLECTION

Objects take up space in memory. If your program creates lots of objects, throws them away, and creates more, you might eventually run out of memory. To reduce the chance that this will happen, Java has garbage collection.

While the Java Virtual Machine (JVM) runs your program, it also spends little bits of time searching for objects that you're no longer using, so it can reclaim their memory for use by other objects.

You don't have to know anything about garbage collection to be an effective Java programmer. But garbage collection is interesting because the JVM uses a lot of hidden data structures to manage memory. These data structures are hidden from your Java program--after all, the JVM, just like any other encapsulated module, should hide the details of how it is implemented. Here's a peak at what's going on under the hood.

Roots and Reachability

Garbage collection's prime directive is to never sweep up an object your program might possibly use or inspect again. These objects are said to be _live_. The opposite of "live" is _garbage_--objects that your program cannot reference again. Java's design makes it possible for the JVM to determine whether an object can ever be used again by your program or not.

Garbage collection begins at the roots. A _root_ is any object reference your program can access directly, without going through another object. There are two kinds (that we know about). First, every local variable (including parameters) in every activation record on the program stack is a root if it is a reference. (Primitive types like ints are not roots; only references are.) Second, every class variable (aka "static" field) that is a reference is a root.

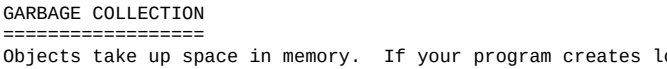
An object is live, and should not be garbage collected, if
- it is referenced by a root (obviously), or
- it is referenced by a field in another live object.

Note that this definition is recursive. Another way to say it is that an object is live if it is _reachable_ from the roots. If you run depth-first search starting at all the roots, you will visit all the live objects and none of the garbage. And that's exactly what garbage collectors do: run depth-first search from the roots.

Each object has a small tag that allows the garbage collector to mark whether the object has been visited or not. The tag is invisible to your Java program, but it takes a few bits of the object's memory. (This is not the only "hidden" memory Java associates with an object--for example, every object has a hidden label that tells Java what class it's in.)

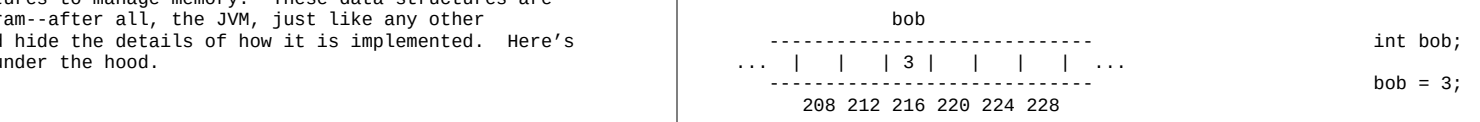
Memory Addresses

In any modern computer, memory is one huge array of bytes with addresses.

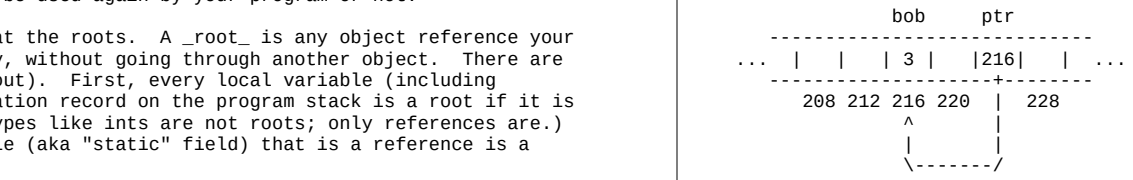


However, modern computers prefer to read or write four bytes at a time, and they do this much faster if the four bytes start at an address divisible by four, so that's how things like ints and floats are stored.

Every time you declare a local variable, you are naming a memory location. (You pick the name, Java picks the address.) An assignment statement writes something into a memory location.



Computers can store memory addresses in memory. To reduce the number of syllables, memory addresses are called _pointers_ for short. Some languages like C allow you to declare variables that are pointers. A pointer field is a memory location that points to another memory location.



Java uses pointers too, but it considers them confidential information, and won't let you print them or look at the numbers directly. Java references are a little bit like pointers, but as we'll learn soon, they're actually more complicated than pointers.

The important point is that your computer's memory is just one giant array that has no structure except the structure you impose on it. Java saves you a huge amount of time and effort by structuring memory for you. Java does this by using hidden pointer-based data structures that you can't access from a Java program.

Mark and Sweep Garbage Collection

A mark-and-sweep garbage collector runs in two separate phases. The _mark_ phase does a DFS from every root, and marks all the live objects. The _sweep_ phase does a pass over all the objects in memory. Each object that was not marked as being live is garbage, and its memory is reclaimed.

How does the sweep phase do a pass over all the objects in memory? The JVM has an elaborate internal data structure for managing the heap. This data structure keeps track of free memory and allocated memory so that new objects can be allocated without overwriting live ones. Time prevents my describing the heap data structure here, though you'll probably implement one in CS 61C. But Java's hidden data structure allows the garbage collector to do a pass over every object, even the ones that are not live. It's roughly like an invisible linked list that links _everything_.

Similarly, the activation records on the stack are data structures that make it possible for the garbage collector to determine which data on the stack are references, and which are not.

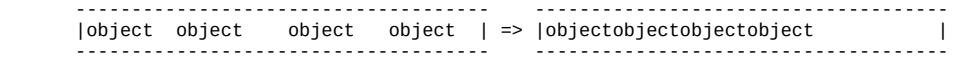
When a mark-and-sweep collector runs, your program stops running for an instant while the garbage collector does a mark pass and a sweep pass. The garbage collector is typically started when the JVM tries to create a new object but doesn't have enough memory for it.

Compaction

Typical programs allocate and forget a good many objects. One problem that arises is _fragmentation_, the tendency of the free memory to get broken up into lots of small pieces. Fragmentation can render Java unable to allocate a large object despite having lots of free memory available.

(Fragmentation also means that the memory caches and virtual memory don't perform as well. If you don't know why, wait until CS 61C or CS 152.)

To solve this problem, a compacting garbage collector actually picks up the objects and moves them to different locations in memory, thereby removing the space between the objects. This is easily done during the sweep phase.

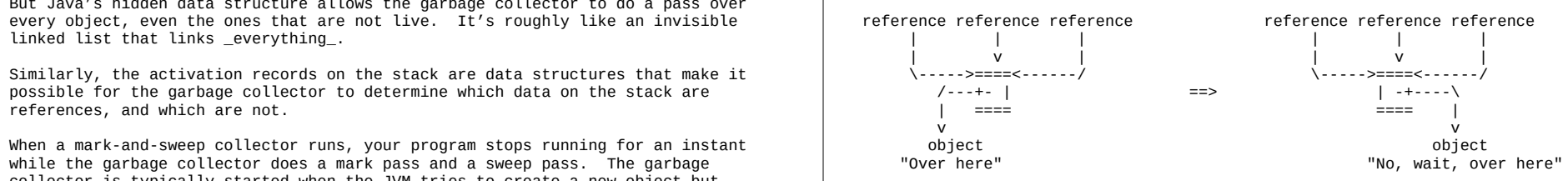


References

There's a problem here: if we pick up an object and move it, what about all the references to that object? Aren't those references wrong now?

Interestingly, in the Sun JVM, a reference isn't a pointer. A reference is a handle. A _handle_ is a pointer to a pointer.

When an object moves, Java corrects the second pointer so it points to the object's new address. That way, even if there are a million references to the object, they're all corrected in one fell swoop. The "second pointers" are kept in a special table, since they don't take as much memory as objects.



Copying Garbage Collection

Copying garbage collection is an alternative to mark and sweep. It does compaction, but it is faster than mark and sweep with compaction because there is only one phase, rather than a mark phase and a sweep phase.

Memory is divided into two distinct spaces, called the old space and the new space. A copying garbage collector finds the live objects by DFS as usual, but when it encounters an object in the old space, it _immediately_ moves it to the new space. The object is moved to the first available memory location in the new space, so compaction is part of the deal. After all the objects are moved to the new space, the garbage objects that remain in the old space are simply forgotten. There is no need for a sweep phase.

Next time the garbage collector runs, the new space is relabeled the "old space" and the old space is relabeled the "new space". Long-lived objects may be copied back and forth between the two spaces many times.

While your program is running (between garbage collections), all your objects are in one space, while the other space sits empty.

The advantage of copying garbage collection is that it's fast. The disadvantage is that you effectively cut in half the amount of heap memory available to your program.