Memory Management

- The process of binding values to memory
- Requires deep understanding of both the dynamic and static characteristics of a program

There are 3 types of memory:
1. Static memory: stores values that are known at compile time, and are constant during execution. Can also store variables with lifetime equal to that of entire program.
2. Runtime stack: stores local variables and other details for function/method calls.
3. Heap: stores everything else (i.e. anything that is dynamically allocated).

Overview of Memory

- A program has an address space in memory, and is restricted to only using space given
- In the example to the right, the program has n+1 bytes available for use:
  - The size of static area is fixed at compile time
  - The top of the stack is address (a−1), which varies as the program executes
  - The start of the heap is at address h, which is fixed at the beginning of execution (determined based upon the upper bound on stack space)
  - The value n determines the total amount of memory, and is fixed at the start of execution
- At runtime, any attempt to use more stack space than given produces a stack overflow
- Any attempt to go beyond the location n produces a heap overflow

Dynamic Data

- **Dynamically** allocated data lives in heap memory
  - The lifetime of heap-allocated-memory extends until the memory is released either by a garbage-collection mechanism (as in Java) or by the program itself (as in C)
  - Heap overflow occurs when we try to allocate new memory during execution and run out of preset space
- **Local** variables are allocated on the stack
  - The lifetime of stack-allocated memory extends until the associated method terminates
  - Stack overflow occurs when the run-time stack hits its limit
Garbage Collection

- In C, memory management is entirely the responsibility of the programmer through manual allocation/deallocation.
- In Java, memory management is entirely the responsibility of the JVM, handled via automatic garbage collection.
- The central issues for the Java model are:
  - How does the JVM identify unreachable objects?
  - How does the JVM free those objects?
  - When does this happen?

Method 1: Reference Counting

- Reference counting is the simplest technique for identifying unreachable objects.
  1. Each object has a reference count associated with it.
  2. The count increases whenever an object is bound to a variable name (this happens during an assignment).
  3. The count decreases whenever an object is unbound (this can also happen during a (re-)assignment).
- The count is not directly accessible to Java programmers (or to Java programs).
- It is manipulated entirely by the JVM, which does some hidden work with every assignment.

A Simple Example

- Consider the code:
  ```java
  Object r1 = new Integer(12);
  Object r2 = r1;
  r1 = null;
  r2 = null;
  if(r2 != r1) {
    if(r2 != null) {
      r2.count--;
    }
    r2 = r1;
    if(r1 != null) {
      r1.count++;
    }
  }
  ```

- JVM might implement assignment `r2 = r1` as if it had code:
  - If objects *already* equal, no change in count needed.
  - If `r2` is non-null, then *decrease* the count of the object it points to, since we are about to de-reference it.
  - After assignment, *increase* count of new object `r2` points to (if that is non-null).

A More Complicated Example

- What about more complex objects?
- Modification of the reference count must *propagate* through the inter-related objects, adjusting counts as necessary while freeing up used memory.

```
public class Pair { // constructor omitted...
    Integer one;
    Integer two;
}
Object r1 = new Pair(new Integer(12), new Integer(21));
r1 = null;
```
Method 2: Mark and Sweep

Whenever heap memory is requested, either:
1. The request is fulfilled because memory is 
   available
2. The request is fulfilled because unreachable objects are 
   freed 
   by mark-and-sweep, resulting in enough memory
3. The request is not fulfilled because there is not enough 
   memory even after mark-and-sweep

Whenever an object is released
- The object is simply taken off runtime stack
- The memory of the object is not released
- Stack is later used to see if memory should be released, when 
  more memory is requested (as above)

Mark-Sweep
- free_list contains all memory locations currently free
- Each node in heap has a mark bit
  - Initially, all nodes are in the free_list, with mark bit = 0
- When heap overflow occurs, we make two sweeps over heap:
  1. Mark all nodes (directly or indirectly) accessible from active 
     variables (i.e., in runtime stack) by setting mark bit = 1
  2. Sweep through the entire heap and return all the unmarked 
     (mark bit = 0) nodes to free_list
- This process returns all orphan objects to free_list

Heap after First Pass of Mark-Sweep
- p and q are still active in run-time 
  stack, so heap space to which they refer is marked with mark bit = 1
- Continues recursively for objects linked to 
  p and q (internal variables, etc.), and for 
  other live variables in run-time stack

When the process is complete, orphaned objects still 
have mark bit = 0, as never reached during sweep
This Week

- **Topic:** Garbage collection; functions and subroutines
- **Reading:** Text, 8.5, 9.1–9.3
- **Meet:**
  - Monday/Wednesday: regular classroom
  - Friday: No class (MICS conference/programming contest)
- **Homework 03:** due Friday, 12 April, 5:00 PM
- **Office Hours:** Wing 210
  - Monday & Wednesday: 9:00 AM – 10:30 AM
  - Tuesday: 3:00 PM – 4:00 PM
  - Thursday: 1:00 PM – 3:00 PM
  - Friday: No office hours this week