Scope, Static, Linked Lists, Arrays

Discussion 02

Announcements

- Weekly Survey 2 - due this Monday 1/29
- Lab 3 - due this Friday 2/2
- Proj 1A - due next Monday 2/5
- Project Party 1/31
- Carefully read the OH guidelines if you attend

Content Review

GRoE: Golden Rule of Equals

```
“Given variables y and x:
y = x copies all the bits from x into y.”
```

Java is pass-by-value: when you call a function and give it some arguments, the function called receives an exact copy of those arguments, tied to its own local variables.

“Copies all the bits” means different things for primitive vs. reference types.

Primitive vs. Reference Types

- Primitive Types are represented by a certain number of bytes stored at the location of the variable in memory. There are only 8 in Java.

  Examples: `byte, short, int, long, float, double, boolean, char`

- Reference Types are represented by a memory address stored at the location of the variable which points to where the full object is (all objects are stored at addresses in memory). This memory address is often referred to as a *pointer*.

  Examples: Strings, Arrays, Linked Lists, Dogs, etc.

Back to the GRoE

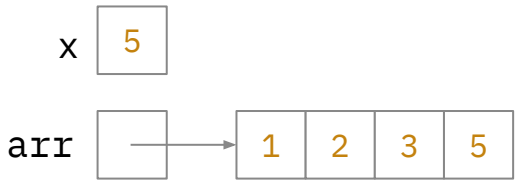
"Given variables y and x:
y = x copies all the bits from x into y."

- The value of a primitive type gets copied directly upon variable assignment
  - Ex. `int x = 5;` means that variable `x` stores the value of 5

- The value of a reference type is a "shallow" copy upon variable assignment: the pointer (memory address) is copied, and the object itself in memory is not
  - Exception: `null` is a special pointer that we compare with `==`

A Quick Example

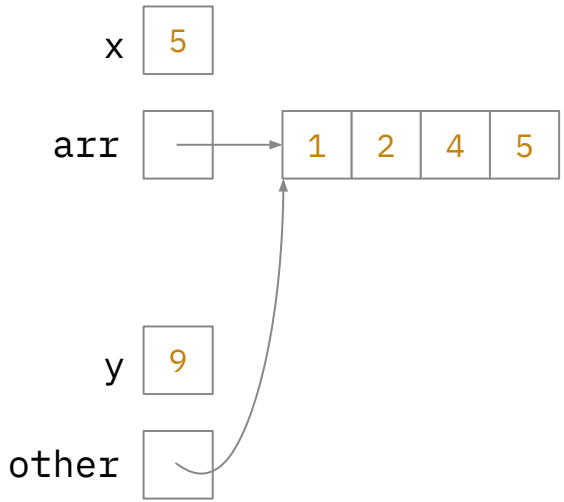
```
int x = 5;
int[] arr = new int[]{1, 2, 3, 5};
```



A Quick Example

```
int x = 5;
int[] arr = new int[]{1, 2, 3, 5};
doSomething(x, arr);
...

public void doSomething(int y, int[] other) {
    y = 9;
    other[2] = 4;
}
```



Static vs. Instance, Revisited

Static variables and functions belong to the whole class.
Example: Every 61B Student shares the same professor, and if the professor were to change it would change for everyone.

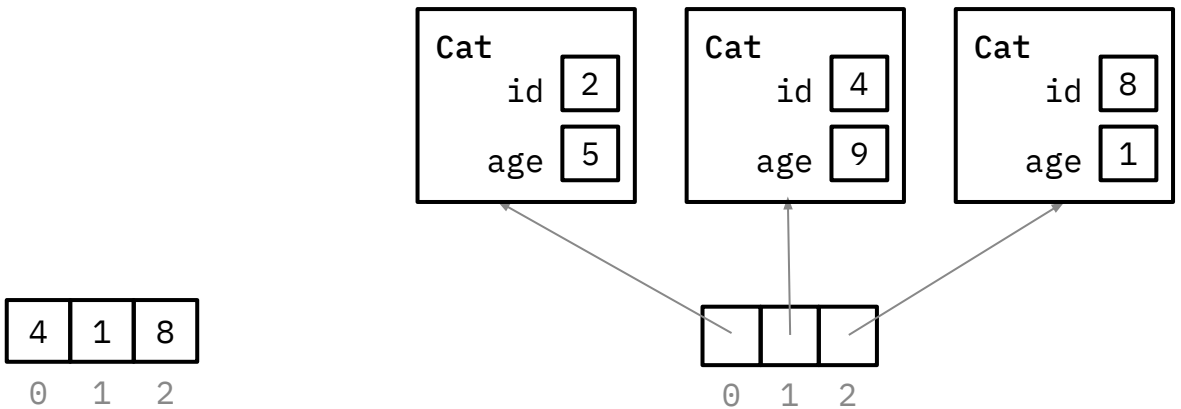
Instance variables and functions belong to each individual instance.
Example: Each 61B Student has their own ID number, and changing a student's ID number doesn't change anything for any other student.

`this` vs. `static`

- `this`
  - Non-static methods can only be called using an instance of that object, so during evaluation of that function, you will always have access to this instance of the object, referred to as `this`
- `static` methods
  - do not require an instance of that object in order to be called, so during evaluation of that function, you cannot rely on access to this instance of the object
- `static` variables
  - shared by all instances of the class; each instance does not get its own copy but can access
- Check for understanding: can you reference `this` in static methods? Can you reference static variables in instance methods? Why or why not?

Arrays

Arrays are data structures that can only hold elements of the same (primitive or reference) type of value. `arr[i]` holds a value in the ith position of the array (zero-indexed). We can also have n-dimensional arrays (ie. `int[][] a = new int[3][2];` you can index into these like `a[2][1]`)



Arrays have a set length when instantiated, so they cannot be extended / shortened with pointers like a Linked List. To resize, we need to copy over all elements to a new array (ie. `System.arraycopy`)

Linked Lists

Linked Lists are modular lists that are made up of nodes that each contain a value and a pointer to the next node. To access values in a Linked List, you must use dot notation.
Example: `intList.get(2)`

- Can be extended or shortened by changing the pointers of its nodes (unlike arrays)
- Can't be indexed directly into like an array: instead, the computer has to iterate through all of the nodes up to that point and follow their next pointers
- A sentinel is a special type of node that is often used as an empty placeholder for ease of adding / deleting nodes, especially from the front or back of the Linked List
    - In a circular doubly-linked implementation, the sentinel's `next` and `prev` pointers are the first and last nodes respectively

Worksheet

CS 61B Spring 2024

Arrays, Linked Lists

Exam-Level 02: January 29, 2024

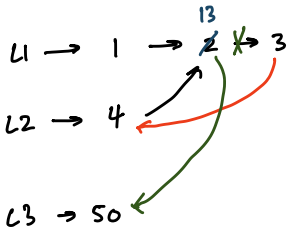
1 Boxes and Pointers

Draw a box and pointer diagram to represent the IntLists `L1`, `L2`, and `L3` after each statement.

```
1  IntList L1 = IntList.list(1, 2, 3);
2  IntList L2 = new IntList(4, L1.rest);
3  L2.rest.first = 13;
4  L1.rest.rest.rest = L2;
5  IntList L3 = IntList.list(50);
6  L2.rest.rest = L3;
```

from null to L2

↳ We lose access to 3!



2 Partition

Implement `partition`, which takes in an `IntList lst` and an integer `k`, and *destructively* partitions `lst` into `k IntLists` with the following properties:

- It is the **same** length as the other lists. You may assume the `IntList` is evenly divisible.
- Its ordering is consistent with the ordering of `lst`, i.e. items in earlier in `lst` must **precede** items that are later.

These lists should be put in an array of length `k`, and this array should be returned.
For instance, if `lst` contains the elements 6, 5, 4, 3, 2, 1, and `k = 2`, then a **possible** partition, is putting elements [6, 4, 2] at index 0, and elements [5, 3, 1] at index 1.

You may assume you have the access to the method `reverse`, which destructively reverses the ordering of a given `IntList` and returns a pointer to the reversed `IntList`. **Hint:** Think about how to build up the IntList backward at each index, starting with `null`.

↳ Doing it forward requires either a pointer to the end or traversal to the end

You may not create any `IntList` instances. → pointers are fine

```
1  public static IntList[] partition(IntList lst, int k) {
2      IntList[] array = new IntList[k];
3      int index = 0;
4      IntList L = reverse(lst)_____________________________________
5      while (L != null) {
6
7          IntList copy = L_____________________________________
8
9          L = L.rest_____________________________________
10
11         IntList old = array[i]_____________________________________
12
13         array[i] = copy_____________________________________
14
15         copy.rest = old_____________________________________
16
17         _____________________________________
18
19         index = index+1_____ % array.length_____;
20     }
21     return array;
22 }
```

Order
4
19
Rest

Example:

lst → 6 → 5 → 4 → 3 → 2 → 1

L → 1 → 2 → 3 → 4 → 5 → 6

copy, L

array

0 → null
1 → null

0 → 1 → null
1 → null

Other orderings are valid, see soln for an alt

3 Remove Duplicates

Using the simplified `DLList` class defined on the next page, implement the `removeDuplicates` method.

`removeDuplicates` should remove all duplicate items from the **DLList**. For example, if our initial list is `[8, 4, 4, 6, 4, 10, 12, 12]`, our final list should be `[8, 4, 6, 10, 12]`. You may **not** assume that duplicate items are grouped together, or that the list is sorted!

8 4 4 6 4 10 12 12

Idea: Since the list is not sorted, go through as though there were two for loops

Order
16
19 43
21 23

```
1  public class DLList {
2      Node sentinel;
3
4      public DLList() {
5          // ...
6      }
7
8      public class Node {
9          int item;
10         Node prev;
11         Node next;
12     }
13
14     public void removeDuplicates() {
15                                          ↓ starting point
16         Node ref = sentinel;
17         Node checker;
18
19         while (ref != null) {
20
21             checker = ref.next;
22
23             while (checker != null) {
24
25 is this a duplicate? → if (ref.item == checker.item) {
26
27                 Node checkerPrev = checker.prev;
28 use to skip "delete" the given node
29                 Node checkerNext = checker.next;
30                 checkerPrev.next = checker.next;
31
32                 checkerNext.prev = checker.prev;
33
34                 ;
35
36                 checker = checkerNext;
37             } else {
38
39                 checker = checker.next;
40             }
41         }
42
43         ref = ref.next;
44     }
45 }
46 }
```