CS 61B Spring 2018

Midterm 1 Guerrilla Section

February 10, 2018

Instructions

Form a small group. Start on the first problem. Check off with a helper or discuss your *solution process* with another group once everyone understands *how to solve* the first problem and then repeat for the second problem ...

You may not move to the next problem until you check off or discuss with another group and *everyone understands why the solution is what it is*. You may use any course resources at your disposal: the purpose of this review session is to have everyone learning together as a group.

1 Take a Knap, hit the sack

1.1 Fix the bugs in `Knapsack` so that `main` prints out `Doge coin : 100.45`.

```
1  class Knapsack {
2      public String thing;
3      public double amount;
4
5      public Knapsack(String str, double amount) {
6          this.thing = str;
7          this.amount = amount;
8      }
9
10     public Knapsack(String str) {
11         this(str, 100.45);
12     }
13
14     public static void main(String[] args) {
15         Knapsack sack = new Knapsack("Doge coin");
16         System.out.println(sack.thing + " : " + sack.amount);
17     }
18 }
```

Meta: Make sure the students understand how constructors work: in order to actually set the instance variables, you need to access them and reassign the ones you created in the fields area. You can't just assign to a new variable in the constructor like in line 6. For the solution's line 6 you technically don't need "this", but in line 7 you do need it because amount has the same name as an instance variable and we need to be able to differentiate between the two. We also want them to know you can only access instance variables by using dot notation on an instance of the class. You can't just access thing and amount like we did in line 16, you need to access it via an instance.

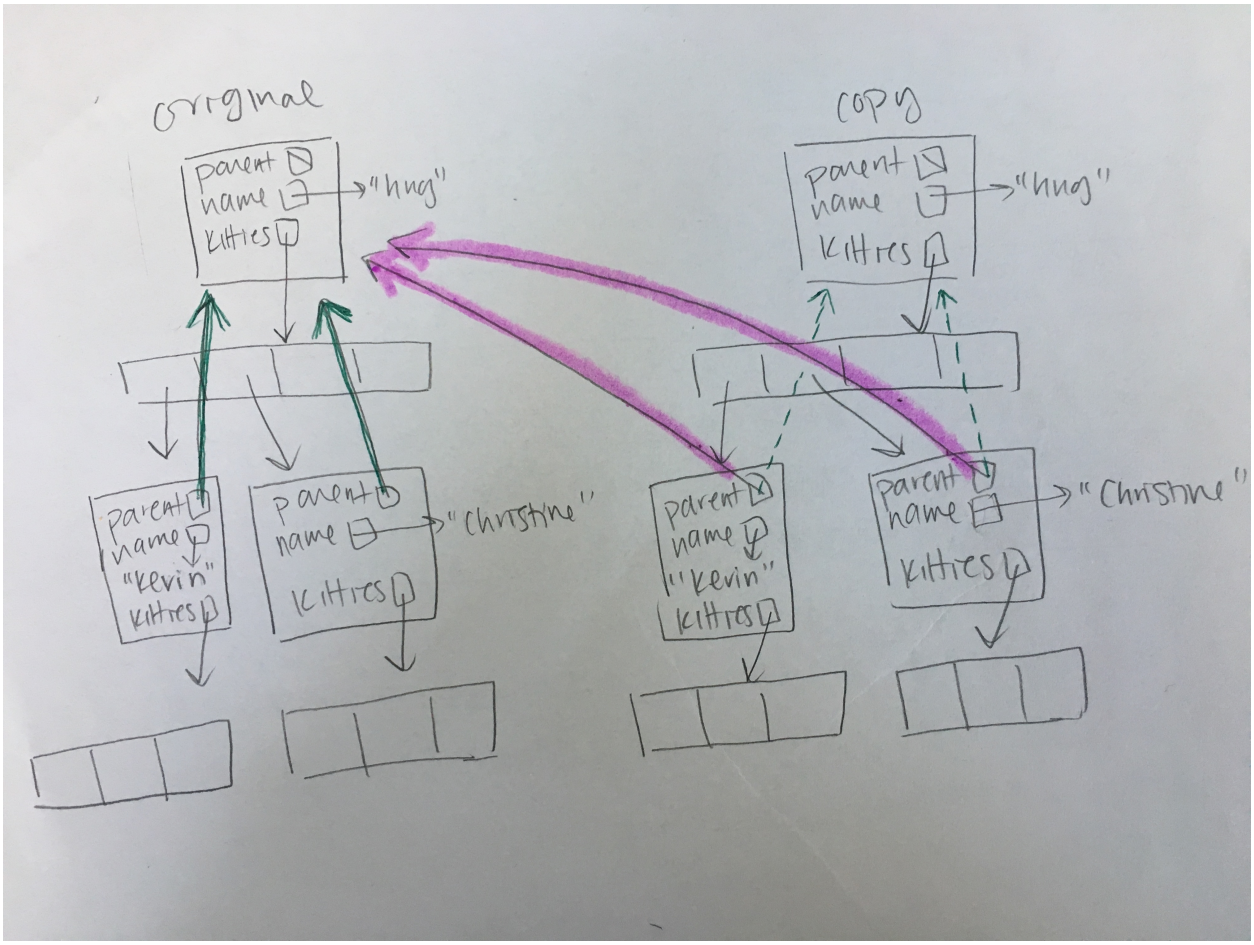
2 I Like Cats

2.1 Toby wants to rule the world with an army of cats. Each cat may or may not have one parent, and may or may not have 'kitties'. Each cat that has a parent is a 'kitty' of that parent. But after implementing `copyCat`, which creates a copy of a cat and its descendants, he realizes the function contains a bug.

```
1  public class Cat {
2      private Cat parent;
3      private ArrayList<Cat> kitties;
4      private String name;
5
6      public Cat(Cat parent, String name) {
7          this.name = name;
8          this.kitties = new ArrayList<Cat>();
9          this.parent = parent;
10     }
11
12     public Cat copyCat() {
13         Cat copy = new Cat(this.parent, this.name);
14         for (int i = 0; i < this.kitties.size(); i += 1) {
15             copy.kitties.add(this.kitties.get(i).copyCat());
16         }
17         return copy;
18     }
19 }
```

What's wrong with his `Cat` class? Drawing a box and pointer diagram may help!

While the parent to child relationships are all correct, the copied child to parent relationships are not. In other words, all of the copied `kitties` are populated correctly, but the `parent` is not — it is never reassigned as it should (dotted green), and thus still points to the old parent (purple).



Meta: We want the students to be able to keep track of what happens to pointers in a recursive function. What things need to be reassigned to make the function work? What things are actually being assigned?

3 Some Sort of Interface

3.1 Suppose we'd like to implement the `SortedList` interface.

```
1  public interface SortedList {
2      /* Return the element at index i, the ith (0-indexed) smallest element. */
3      int get(int i);
4
5      /* Remove the element at index i, the ith (0-indexed) smallest element. */
6      int remove(int i);
7
8      /* Insert an element into the SortedList, maintaining sortedness. */
9      void insert(int elem);
10
11     /* Return the size of the SortedList. */
12     int size();
13 }
```

(a) Suppose we'd like to optimize the speed of `SortedList::get`. Should we implement `SortedList` with a linked list or an internal array?

An array, since array access can be constant time.

(b) Implement the default method, `merge`, which takes another `SortedList` and merges the other values into the current `SortedList`.

```
default void merge(SortedList other) {
    for (int i = 0; i < other.size(); i += 1) {
        this.insert(other.get(i));
    }
}
```

Meta: The point of this exercise is we want to make use of the methods already given to us. We implement this interface, which means we must have implementations for each of the methods above. This means we can assume that we have `.insert()` and `.get()`.

(c) Suppose we'd like to `merge` using only a constant amount of additional memory. Should we implement `SortedList` with a linked list or an internal array?

Linked list. This way, we can modify the pointers instead of allocating larger and larger arrays to fit all the items in. With an array, we would have to resize to double the length every time, which is not constant additional memory. `other`.

(d) Implement the default method, `negate`, which destructively negates all the values in the `SortedList`.

```
default void negate() {
    if (size() == 1) {
        insert(-remove(0));
    } else if (size() > 1) {
        int smallest = remove(0);
        negate();
        insert(-smallest);
    }
}
```

Meta: Again, the trick is to use the methods in the interface, because we know we must have implemented those no matter what. Handling negative numbers requires some additional care. Other approaches using data structures will also work.

4 Arrayana Grande

4.1 After executing the code, what are the values of `Foo` in `xx` and `yy`?

```
1  public class Foo {
2      public int x, y;
3
4      public static void main(String[] args) {
5          int N = 3;
6          Foo[] xx = new Foo[N], yy = new Foo[N];
7          for (int i = 0; i < N; i++) {
8              Foo f = new Foo();
9              f.x = i; f.y = i;
10             xx[i] = f;
11             yy[i] = f;
12         }
13         for (int i = 0; i < N; i++) {
14             xx[i].y *= 2;
15             yy[i].x *= 3;
16         }
17     }
18 }
```

(a) `xx[0]`

`0 0`

(b) `xx[1]`

`3 2`

(c) `xx[2]`

`6 4`

(d) `yy[0]`

`0 0`

(e) `yy[1]`

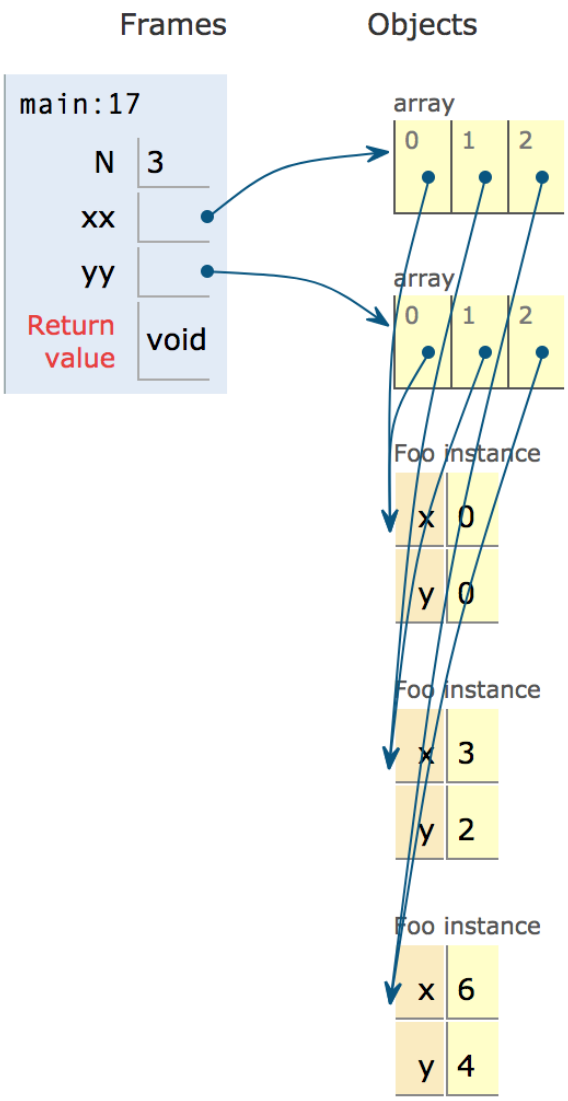
`3 2`

(f) `yy[2]`

`6 4`

Environment diagram below.

Meta: Teach students about pass-by-value and show them how that is reflected in box and pointer diagrams. Encourage them to draw a box and pointer diagram. It will help a lot.



5 Just another fiendly...

5.1 What is the output after running the `main` method in the `Ghoul` class?

```java
public class Monster {
    public String noise = "blargh";
    public static int spookFactor = 5;

    public Monster() {
        System.out.println("Muhahaha!!!");
    }

    public void spook(Monster m) {
        System.out.println("I go " + noise);
    }

    public void spook(Ghoul g) {
        System.out.println("I am " + spookFactor + " spooky.");
    }
}
```

```java
public class Ghoul extends Monster {
    public Ghoul() {
        System.out.println("I am a ghoul");
    }

    public void spook(Ghoul g) {
        System.out.println("I'm so ghoul");
        ((Monster) g).spook(g);
    }

    public void haunt() {
        Monster m = this;
        System.out.println(noise);
        m.spook(this);
    }

    public static void main(String[] args) {
        Monster m = new Monster();
        m.spook(m);

        Monster g = new Ghoul();
        g.spook(m);
        g.spook(g);

        Monster.spookFactor = 10;
        Ghoul ghastly = new Ghoul();
        ghastly.haunt();
    }
}
```

```
Muhahaha!!!  // line 18, call Monster's constructor.
I go blargh  // line 19, call Monster's spook method with Monster's noise.
Muhahaha!!!  // Always need to call superclass' constructors first! This goes for every superclass in
     the hierarchy.
I am a ghoul // After calling superclass's constructor, calls own constructor
I go blargh  // Will call Monster's spook method because the parameter, m's, static is a moster and
    only Monster's spook method takes in a Monster.
I go blargh  // Will again call Monster's spook method because g's static type is monster.
Muhahaha!!!  // Caused by line 26, need to call Monster constructor first
I am a ghoul // Also line 26, after calling monster's constructor call Ghoul's constructor
blargh       // Caused by line 13, ghoul inherits monster's noise
I'm so ghoul // line 14 calls line 7 because we are calling spook method that takes in a Ghoul (this'
    s static type is Ghoul). The subclass Ghoul overrode Monster's spook, so we will call spook from
    Ghoul's class.
I'm so ghoul // line 7
I'm so ghoul // line 7
I'm so ghoul
I'm so ghoul
...
StackOverflowError // We are calling Ghoul's spook recursively ad infinitum.
```

6 David HasselHoF

```java
public interface BinaryFunction {
    public int apply(int x, int y);
}
```

```java
public interface UnaryFunction {
    public int apply(int x);
}
```

6.1 Implement `Adder`, which implements the `BinaryFunction` interface and adds two numbers together.

```java
public class Adder implements BinaryFunction {
    int apply(int x, int y) {
        return x + y;
    }
}
```

Meta: A binary function is just a function that takes two arguments and operates on both of them. This problem checks if students know what interfaces are and how to implement them.

6.2 Implement `Add10` which implements `UnaryFunction`. Its `apply` method returns `x + 10` without using any of the `+ - * /` operators.

```java
public class Add10 implements UnaryFunction {
    private static final Adder add = new Adder();

    int apply(int x) {
        return add.apply(x, 10);
    }
}
```

Meta: A Unary function is a function that takes in one argument and operates on it. Again, we are checking if they know how to implement interfaces. We want them to think about how to perform operations using objects/functions, which in this case requires them creating a new Adder object and calling its apply method.

6.3 Implement `Multiplier` which implements `BinaryFunction`. Its `apply` method accepts two integers, `x` and `y`, and return `x * y` without using any of the `+ - * /` operators except to increment indices in loops. Assume all inputs are positive.

```
1  public class Multiplier implements BinaryFunction {
2      private static final Adder add = new Adder();
3
4      public int apply(int x, int y) {
5          int result = 0;
6          for (int i = 0; i < y; i++) {
7              result = add.apply(result, x);
8          }
9          return result;
10     }
11 }
```

Meta: Again, we want them to reuse the Adder's apply function in order to teach them about HoF's.

7 Tri another angle *Extra*

7.1 Implement `triangularize`, which takes an `IntList[] R` and a single `IntList L`, and breaks `L` into smaller `IntLists`, storing them into `R`.

The `IntList` at index `k` of `R` has at most `k + 1` elements of `L`, in order. Thus concatenating all of the `IntLists` in `R` together in order would give us `L` back.

Assume `R` is big enough to do this. For example, if the original `L` contains `[1, 2, 3, 4, 5, 6, 7]`, and `R` has 6 elements, then on return `R` contains `[[1], [2, 3], [4, 5, 6], [7], [], []]`. If `R` had only 2 elements, then on return it would contain `[[1], [2, 3]]`. `triangularize` may destroy the original contents of the `IntList` objects in `L`, but does not create any new `IntList` objects.

Note: Assume `R`'s items are all initially **`null`**.

```
1  public static void triangularize(IntList[] R, IntList L) {
2      int i, k;                // i is the index of R we are currently working with
3      i = 0; k = 0;            // k is the number of links left to put in R[i]
4      while (i < R.length) {   // terminate when we get to the end of R because we can only insert as
       many links as R can hold
5          if (k == 0) {        // if we haven't inserted any links yet
6              R[i] = L;        // set R[i] to the beginning of L
7          }
8          if (L == null) {     // if there is no list left, we will just run the function until we
       reach the false case for the while loop... although we could just return...
9              i += 1;
10             k = 0;
11         } else if (k == i) { // We have put the correct number of links into this index
12             IntList next = L.rest; // save pointer to next link
13             L.rest = null;   // set list's rest to null to terminate the list
14             L = next;        // reset L to the saved pointer so we can continue in next bucket
15             i += 1;          // move i to next index
16             k = 0;           // reset k
17         } else {             // We are just gonna insert the links
18             L = L.rest;      // move forward one link
19             k += 1;          // increment k because we just inserted one link.
20         }
21     }
22 }
```

Meta: We want the students to know how to traverse through and manipulate linked lists. Drawing environment diagrams really helps. Walk them through the steps on paper, and then facilitate translation of those steps into code.

8 Arrays of the 2D variety *Extra*

8.1 Implement `diagonalFlip`, a method that takes a 2-D array `arr` of size $N \times N$ and **destructively** flips `arr` along the diagonal line from the left bottom to right top.

```
1  public static void diagonalFlip(int[][] arr) {
2      int N = arr.length;
3      for (int i = N-1; i >= 0; i--) {
4          for (int j = 0; j < N-i-1; j++) {
5              int temp = arr[i][j];
6              arr[i][j] = arr[N-j-1][N-i-1];
7              arr[N-j-1][N-i-1] = temp;
8          }
9      }
10 }
```

The intuition behind this is that you need to swap all of the items, two-by-two, along the diagonal. To visualize it, say you have a 5×5 array:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Then you want to swap

```
1 2 3 4 5    with     5
1 2 3 4              4 5
1 2 3              3 4 5
1 2              2 3 4 5
1              1 2 3 4 5
```

The middle diagonal will just stay the same. We know that we need a nested for loop, because we have 2 variables to keep track of: row and column. We will have one for loop for keeping track of columns, then another for the rows. The idea is that we will go through the top triangle and then use arithmetic to turn the indices of the item we are looking at to the indices of its swapping partner in the other.

Let's go through a few example swaps: `arr[4][0]` (1) needs to be swapped with `arr[0][4]` (5), `arr[3][2]` (3) needs to be swapped with `arr[1][2]` (3), `arr[4][3]` (4) needs to be swapped with `arr[0][1]` (2). We see a pattern: if the original index is (i, j), then the swapping item's index is $[5-1-j][5-1-i]$ where the 5 is the length of the array and $5-1$ is necessary because of the zero-indexed nature of arrays. We can generalize this to any N-length array: $(i, j) \rightarrow [N-1-j][N-1-i]$. The big picture is that diagonally flipping an array is basically making the columns rows and the rows columns! which is why `arr[i][j]` should be swapped with `arr[some operation on j][some operation on i]`. Now, recall that in order to swap two items in an array, you must have a temp variable. Thus, inside the for loops, you must first declare a temp variable `arr[i][j]` to store that value.

8.2 Implement `rotate`, which takes a 2-D array `arr` of size $N \times N$ and **destructively** rotates `arr` 90 degrees clockwise.

```
public static void rotate(int[][] arr) {
    int N = arr.length;
    for (int i = 0; i < N/2; i++) {
        for (int j = i; i < N - j - 1; j++) {
            int temp = arr[i][j];
            arr[i][j] = arr[N-(j+1)][i];
            arr[N-(j+1)][i] = arr[N-i-1][N-(j+1)];
            arr[N-i-1][N-(j+1)] = arr[j][N-i-1];
            arr[j][N-i-1] = temp;
        }
    }
}
```