

# CS 61B Spring 2017 Guerrilla Section 2 Solutions

12 February 2017

Directions: In groups of 4-5, work on the following exercises. Do not proceed to the next exercise until everyone in your group has the answer and *understands why the answer is what it is*. Of course, a topic appearing on this worksheet does not imply that the topic will appear on the midterm, nor does a topic not appearing on this worksheet imply that the topic will not appear on the midterm.

## 1 ADT Selection

Suppose we'd like to implement the `SortedList` interface. It's API looks like this:

```
1 public interface SortedList {
2     /* Initialize a SortedList with one element. */
3     public SortedList(int elem);
4
5     /* Get the element at index i. */
6     public int get(int i);
7
8     /* Merge this list with other. Postcondition: this SortedList must remain in sorted
9         order */
10    public void merge(SortedList other);
11 }
```

1. Suppose we'd like to perform merge operations between lists using only a constant amount of additional memory. Should `SortedList` be implemented using an internal linked list or an internal array?  
**Linked list.** This would allow us to easily insert elements to a list during the merging process, by setting pointers between the nodes.
  
2. Now suppose we'd like to optimize the speed of our `SortedList` data structure's get operations. Again, select an internal data structure (array or linked list) for `SortedList`.  
**Array.** With an array as the internal data structure, the get method can be implemented in constant time.

# STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

## 2 ALists

Here's an incomplete implementation of the AList class.

1. Implement the `delete(int index)` method, which deletes the array element at index `i` and shifts the remaining elements of the array up. You may assume that `i` is between 0 and `size - 1`, inclusive. If `delete` causes the load on items to be less than .25 (that is, if items becomes less than a quarter full), `resize` items to be half its current capacity.
2. Write the integer that would be printed on the line next to the `System.out.println(...)` methods in the main method.

```
1 public class AList {
2     public int size;
3     public int[] items;
4
5     public AList() {
6         size = 0;
7         items = new int[2];
8     }
9
10    /** Capacity doubles whenever the size exceeds the capacity. */
11    public void insertBack(int x) {
12    }
13
14    public int getBack() {
15    }
16
17    public int deleteBack() {
18    }
19
20    public int get(int index) {
21    }
22
23    public int capacity() {
24        return items.length;
25    }
26
27    /** This will be used by insertBack(int x), deleteBack() and delete(). */
28    public void resize(int newCapacity) {
29    }
30
31    public int delete(int index) {
32        /* Your implementation. */
33        if (index == size - 1) {
34            return deleteBack();
35        } else {
36            int item = items[index];
37            for (int i = index; i < size - 1; i++) {
38                items[i] = items[i + 1];
39            }
40
41            size = size - 1;
42            if (size < capacity() / 4) {
43                resize(capacity() / 2);
44            }
45            return item;
46        }
47    }
48
49    public void main(String[] args) {
50        AList alist = new AList();
51        for (int i = 0; i < 5; i++) {
52            alist.insertBack(i);
53        }
54    }
55 }
```

```
48     }
49     System.out.println(alist.size);           __5__
50     System.out.println(alist.capacity());    __8__
51     alist.deleteBack();
52     alist.deleteBack();
53     System.out.println(alist.capacity());    __8__
54     alist.deleteBack();
55     alist.deleteBack();
56     System.out.println(alist.size);         __1__
57     System.out.println(alist.capacity());   __4__
58 }
59 }
```

# STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

### 3 Abstract classes

1. What are advantages and disadvantages of abstract classes and interfaces?

Abstract classes allow you create a partially defined class with abstract methods – you can define some methods, while leaving abstract methods as method headers for subclasses to implement. Interfaces only allow method headers to be defined, without implementation. However, abstract classes cannot be instantiated, because they may contain unimplemented methods.

EDIT: Interfaces now have default methods. Additionally, a class can implement multiple interfaces but only extend one abstract class.

2. Consider this school class:

```
1 public class School {
2     String name;
3     int numStudents;
4
5     public void cheer() {
6         System.out.println("I have no idea what to say.");
7     }
8
9     public void enrollStudent() {
10        numStudents += 1;
11        if (numStudents % 1000 == 0) {
12            System.out.println("We have " + numStudents + " students!");
13        }
14    }
15
16    public void expelStudent() {
17        students -= 1;
18    }
19 }
```

Enrolling and expelling students makes sense but we don't know what a School should do for its cheer. We intend to make subclasses of schools that have their own special way of cheering. How should we rewrite school? Mark changes on the prewritten class above.

Make the class abstract and change cheer method to abstract.

3. We want to create a University class so we can create school instances of different education levels. Oski tried his best, but he didn't take CS61B. University cheers should output the name followed by a space and the motto. Also, Oski forgot that Universities congratulate students upon enrolling them. In addition to doing what enroll currently does, the method should also print "Congratulations!". Fix Oski's University class so it compiles and follows University behaviors. (Try to add as few lines as possible. Feel free to cross things out.)

```

1 public class University extends School {
2     String motto; // they should add this. but not add name.
3
4     public University(String name, String motto) {
5         this.name = name;
6         this.motto = motto;
7     }
8
9     public void cheer() {
10        // change this to void and removed the return statement.
11        String chant = name + ' ' + motto;
12        System.out.println(chant);
13
14    }
15
16    public void enrollStudent() {
17        // Should ask students to go in this direction instead of copyasting the
18        // original method
19        super.enrollStudent();
20        System.out.println("Congratulations!")
21    }
22 }

```

4. Stanford thinks they are too cool for school. They wrote their own class following University guidelines. But it's quite unnecessary.

```

1 public class Stanford {
2     public void cheer() {
3         System.out.println("Stanfurd is 2cool4skool");
4     }
5
6     public void enrollStudent() {
7         numStudents += 1;
8         if (numStudents % 1000 = 0) {
9             System.out.println("We have " + numStudents + " students!");
10        }
11        System.out.println("Congratulations!")
12    }
13
14    public void expelStudent() {
15        students -= 1;
16    }
17 }

```

Show how simple it is to create an School instance of Stanford with the same functionality.

```
School Stanford = new University ("Stanfurd", "is 2cool4skool");
```

## STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

## 4 What Would Java Display?

Consider the following classes. What is the output after running the main method in the Monster class?

```
1 class Ghoul extends Monster {
2     public Ghoul() {
3         System.out.println("I am a ghoul.");
4     }
5
6     public void spook() {
7         System.out.println("I'm so ghoul: " + noise);
8         System.out.println("I am " + spookFactor + " spooky.");
9     }
10
11     public static void mash(Ghoul g) {
12         System.out.println("boogity boo: ");
13         g.spook();
14         //spook();
15     }
16
17     public void haunt() {
18         System.out.println("ERRERERRRRERRR");
19         mash(this);
20     }
21 }
22
23 public class Monster {
24     protected String noise = "blargh";
25     public static int spookFactor = 5;
26
27     public Monster() {
28         System.out.println("Muhahaha!!!");
29     }
30
31     public void spook() {
32         System.out.println("I go " + noise);
33         System.out.println("I am " + spookFactor + " spooky.");
34     }
35
36     public static void mash(Monster m) {
37         System.out.println("Monster: ");
38         m.spook();
39     }
40
41     public static void main(String[] args) {
42         // part a
43         System.out.println("Part a:");
44         Monster m = new Monster();
45         m.mash(m);
46
47         System.out.println("Part b:");
48         Monster g = new Ghoul();
49         g.mash(g);
50
51         System.out.println("Part c:");
52
53         g.spookFactor = 10;
54         m.mash(m);
55     }
```

```
56     System.out.println("Part d:");
57
58     Ghoul ghastrly = new Ghoul();
59     m = ghastrly;
60     ghastrly = (Ghoul) m;
61     ghastrly.haunt();
62     m.mash(ghastrly);
63 }
64 }
```

Part a:

Muhahaha!!!

Monster:

I go blargh

I am 5 spooky.

Part b:

Muhahaha!!!

I am a ghoul.

Monster:

I'm so ghoul: blargh

I am 5 spooky.

Part c:

Monster:

I go blargh

I am 10 spooky.

Part d:

Muhahaha!!!

I am a ghoul.

ERRRERERRRRERRR

boogity boo:

I'm so ghoul: blargh

I am 10 spooky.

Monster:

I'm so ghoul: blargh

I am 10 spooky.

# STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

## 5 Vehicle Interfaces

Henry Hacker wrote up two Vehicle classes Car and Plane. He notices that most if not all Vehicles will need to check their lights and gas for a maintenance report. He also wants to enforce having all Vehicles be able to check lights and gas as well as provide a maintenance report. Below, write an interface for Vehicle

```
1 public interface Vehicle {
2     public void checkGas();
3     public void checkLights();
4     default public void reportMaintenance() {
5         checkGas();
6         checkLight();
7     }
8 }
9
10 public class Car implements Vehicle{
11     private int gas;
12     private boolean lightsWork;
13     public void honk() {
14         System.out.println("Honk!");
15     }
16     public void checkLights() {
17         if (!lightsWork) {
18             System.out.println("Lights are broken.");
19         }
20     }
21     public void checkGas() {
22         if (gas < 3) {
23             System.out.println("Need more gas.");
24         }
25     }
26     public void reportMaintenance() {
27         checkGas();
28         checkLight();
29     }
30 }
31 public class Plane implements Vehicle {
32     private int gas;
33     private int battery;
34     public void checkGas() {
35         if (gas < 1000) {
36             System.out.println("Should refuel before flying.");
37         }
38     }
39     public void checkLights() {
40         if (battery < 10) {
41             System.out.println("Should replace battery for lights.");
42         }
43     }
44     public void reportMaintenance() {
45         checkGas();
46         checkLight();
47     }
48 }
```



## 6 Inheritance Bonus

(From Fall 2014, Midterm 1.)

Fill in the blanks and cross out and rewrite lines of code in the Animal and Dog classes so that Foo.java compiles and prints out the following lines:

```
1
2
3
Superdog
Superdog
bark 3
4
```

Do not cross out a line and replace it with multiple lines (i.e. just rewrite the line you cross out). Do not add new lines of code anywhere except the blanks provided. Do not modify Foo. Do not modify any lines that say "do not modify"

```
1 import zoo.Animal;
2 import housepets.Dog;
3 public class Foo { // Do not modify class Foo
4
5     public static void main(String[] args) {
6         Animal a = new Dog();
7         Animal b = new Dog();
8         Animal c = new Dog();
9         a.makeNoise();
10        b.makeNoise();
11        a.makeNoise();
12        c.sayName();
13        a.sayName();
14        a.makeNoise("bark");
15        c.makeNoise();
16    }
17 }
```

```
1 package zoo;
2 public class Animal {
3
4     protected static int noise; // CHANGED FROM int noise;
5     private String name; // Do not modify this line
6
7     public Animal(String name) {
8         this.name = name; // CHANGED FROM name = name;
9     }
10
11    public void makeNoise() {
12        noise += 1;
13        System.out.println(noise);
14    }
15
16    public void sayName() {
17        System.out.println(name);
18    }
19
20    public void makeNoise(String sound){}
21 }
```

```
1 package housepets;
2 import zoo.Animal;
3 public class Dog extends Animal { // CHANGED FROM public class Dog {
4
5     public Dog() {
6         super("Superdog");
7     }
8
9     public void makeNoise(String sound) {
10        System.out.println(sound + " " + noise);
11    }
12 }
```

# STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

## 7 HOF

Use these two interfaces for the following problems.

```

1 public interface BinaryFunction {
2     public int apply(int x, int y);
3 }
4
5 public interface UnaryFunction {
6     public int apply(int x);
7 }

```

1. Implement the Adder class below, which should implement the BinaryFunction interface and add two numbers together.

```

1 public class Adder implements BinaryFunction {
2     int apply(int x, int y) {
3         return x + y;
4     }
5 }

```

2. Implement the Add10 class. It should have a single method ?apply? that takes in a single integer x and returns x + 10 without using any of the +-\* / operators.

```

1 public class Add10 implements UnaryFunction {
2
3     private static final Adder add = new Adder();
4
5     int apply(int x) {
6         return add.apply(x, 10);
7     }
8 }

```

3. Finish the implementation of the AddX0 class. It should take in an integer to its constructor and its apply method should add 10 times that integer to whatever is passed in without using any of the +-\* / operators (except to increment indices in for/while loops).

```

1 public class AddX0 implements UnaryFunction {
2
3     private static final Add10 addTen = new Add10();
4     // not necessary to init here but recommended
5     private int tensPlace;
6
7     public AddX0(int num) {
8         tensPlace = num;
9     }
10
11     public int apply (int x) {
12         int result = x;
13         for (int i = 0; i < tensPlace; i++) {
14             result = addTen.apply(result);
15         }
16     }
17 }

```

4. Fill in the implementation of the Multiplier class below. Its apply method should take in two ints (x and y) and return x \* y without using any of the +-\* / operators (except to increment indices in for/while loops). You may assume that all inputs are positive.

```
1 public class Multiplier implements BinaryFunction {
2
3     private static final Adder add = new Adder();
4     // not necessary to init here but recommended
5
6     public int apply(int x, int y) {
7         int result = 0;
8         for (int i = 0; i < y; i++) {
9             result = add.apply(result, x);
10        }
11        return result;
12    }
13 }
```

5. Bonus: Rewrite the apply method of the Multiplier class below to take negative inputs into account. You still may not use the `+*/` operators, with the exception that you may use the unary version of the `-` operator to negate numbers.

```
1 public int apply(int x, int y) {
2     if (y < 0) {
3         return -apply(x, -y);
4     } else {
5         int result = 0;
6         for (int i = 0; i < y; i++) {
7             result = add.apply(result, x);
8         }
9         return result;
10    }
11 }
```

# STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!