# CS61B Spring 2016 Guerrilla Section 1b Worksheet SOLUTIONS

Akhil Batra, Leo Colobong, Nick Fong, Laura Harker, Anusha Ramakuri, Giulio Zhou, Daniel Socher, Mitas Ray, Kuriakose Sony Theakanath, Dasheng Chen, Jimmy Lee, Bhuvana Bellala, Matthew Mussomele, Andy Zhang

7 February 2016

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 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.

```
public class AList {
  public int size;
  public int[] items;

  public AList() {
    size = 0;
    items = new int[2];
  }

  /** Capacity doubles whenever the size exceeds the capacity. */
  public void insertBack(int x) {     }

  public int getBack() {     }

  public int deleteBack() {     }

  public int get(int index) {     }

  public int capacity() {
    return items.length;
  }

  /** This will be used by insertBack(int x), deleteBack() and delete(). */
  public void resize(int newCapacity) {     }

  public int delete(int index) {
    /* Your implementation. */
    if (index == size - 1) {
      return deleteBack();
    } else {
      int item = items[index];
      for (int i = index; i < size - 1; i++) {
        items[index] = items[index + 1];
      }

      size = size - 1;
      if (size < capacity() / 4) {
        resize(capacity() / 2);
      }
      return item;
    }
  }

  public void main(String[] args) {
    AList alist = new AList();
    for (int i = 0; i < 5; i++) {
      alist.insertBack(i);
    }
    System.out.println(alist.size);              __5__
    System.out.println(alist.capacity());        __8__
    alist.deleteBack();
    alist.deleteBack();
    System.out.println(alist.capacity());        __8__
    alist.deleteBack();
    alist.deleteBack();
    System.out.println(alist.size);              __1__
    System.out.println(alist.capacity());        __4__
  }
}
```

STOP!

DON'T PROCEED UNTIL EVERYONE IN YOUR GROUP HAS FINISHED AND UNDERSTANDS ALL EXERCISES IN THIS SECTION!

## 2 ADT Selection

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

```java
public interface SortedList {
  /* Initialize a SortedList with one element. */
  public SortedList(int elem);

  /* Get the element at index i. */
  public int get(int i);

  /* Merge this list with other. Postcondition: this SortedList must remain in
      sorted order */
  public void merge(SortedList other);
}
```

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 stucture'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!

## 3 OOP

```java
1  abstract class Abstraction {
2    abstract void foo();
3  }
4
5  class Bar {
6    void foo() {
7        System.out.println("Berkeley!");
8    }
9  }
10
11 public class AbstractExample {
12   public static void callFoo(Abstraction widget) {
13       widget.foo();
14   }
15
16   public static void main (String[] args) {
17       Object theBar = new Bar();
18       callFoo((Abstraction) theBar);
19   }
20 }
```

For the code above, answer the following questions:

1. Does this code compile? If not, what's the compile-time error?
   The code compiles.

2. Does this code run? If not, what's the run-time error?
   The code does not run. ClassCastException – Bar can?t be casted to Abstraction because an object of class Bar is not a subclass of Abstraction, despite having the same methods

3. If this code does not compile/run, what is the minimum change needed to print "Berkeley!"?
   Change line 5 to read class "Bar extends Abstraction{"

STOP!

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

## 4 What would Java Print?

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

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

  public void spook() {
    System.out.println("I'm so ghoul: " + noise);
    System.out.println("I am " + spookFactor + " spooky.");
  }

  public static void mash(Ghoul g) {
    System.out.println("boogity boo: ");
    g.spook();
    //spook();
  }

  public void haunt() {
    System.out.println("ERRERERRRRERRR");
    mash(this);
  }
}

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

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

  public void spook() {
    System.out.println("I go " + noise);
    System.out.println("I am " + spookFactor + " spooky.");
  }

  public static void mash(Monster m) {
    System.out.println("Monster: ");
    m.spook();
  }

  public static void main(String[] args) {
    // part a
    System.out.println("Part a:");
    Monster m = new Monster();
    m.mash(m);

    System.out.println("Part b:");
    Monster g = new Ghoul();
    g.mash(g);

    System.out.println("Part c:");

    g.spookFactor = 10;
    m.mash(m);

    System.out.println("Part d:");

    Ghoul ghastly = new Ghoul();
    m = ghastly;
    ghastly = (Ghoul) m;
    ghastly.haunt();
    m.mash(ghastly);
  }
}
```

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.
ERRERERRRRERRR
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 maintainence report. He also wants to enforce having all Vehicles be able to check lights and gas as well as provide a maintainence report. Below, write an interface for Vehicle

```java
public interface Vehicle {
    public void checkGas();
    public void checkLights();
    default public void reportMaintainence() {
        checkGas();
        checkLight();
    }
}

public class Car implements Vehicle{
    private int gas;
    private boolean lightsWork;
    public void honk() {
        System.out.println("Honk!");
    }
    public void checkLights() {
        if (!lightsWork) {
            System.out.println("Lights are broken.");
        }
    }
    public void checkGas() {
        if (gas < 3) {
            System.out.println("Need more gas.");
        }
    }
    public void reportMaintenence() {
        checkGas();
        checkLight();
    }
}
public class Plane implements Vehicle {
    private int gas;
    private int battery;
    public void checkGas() {
        if (gas < 1000) {
            System.out.println("Should refuel before flying.");
        }
    }
    public void checkLights() {
        if (battery < 10) {
            System.out.println("Should replace battery for lights.");
        }
    }
    public void reportMaintenence() {
        checkGas();
        checkLight();
    }
}
```

## 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”

```java
import zoo.Animal;
import housepets.Dog;
public class Foo { // Do not modify class Foo

    public static void main(String[] args) {
        Animal a = new Dog();
        Animal b = new Dog();
        Animal c = new Dog();
        a.makeNoise();
        b.makeNoise();
        a.makeNoise();
        c.sayName();
        a.sayName();
        a.makeNoise("bark");
        c.makeNoise();
    }
}
```

```java
package zoo;
public class Animal {

    protected static int noise;  // CHANGED FROM int noise;
    private String name; // Do not modify this line

    public Animal(String name) {
        this.name = name;  // CHANGED FROM name = name;
    }

    public void makeNoise() {
        noise += 1;
        System.out.println(noise);
    }

    public void sayName() {
        System.out.println(name);
    }

    public void makeNoise(String sound){}
}
```

```java
package housepets;
import zoo.Animal;
public class Dog extends Animal {  // CHANGED FROM public class Dog {

    public Dog() {
        super("Superdog");
    }

    public void makeNoise(String sound) {
        System.out.println(sound + " " + noise);
    }
}
```

STOP!

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

## 7 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:

```java
public class School {
    String name;
    int numStudents;

    public void cheer() {
        System.out.println("I have no idea what to say.");
    }

    public void enrollStudent() {
        numStudents += 1;
        if (numStudents % 1000 == 0) {
            System.out.println("We have " + numStudents + " students!");
        }
    }

    public void expelStudent() {
        students -= 1;
    }
}
```

   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.)

```java
public class University extends School {
    String motto;  // they should add this. but not add name.

    public University(String name, String motto) {
        this.name = name;
        this.motto = motto;
    }

    public void String cheer() {
        // change this to void and removed the return statement.
        String chant = name + ' ' + motto;
        System.out.println(chant);

    }

    public void enrollStudent() {
        // Should ask students to go in this direction instead of copypasting
           the original method
        super.enrollStudent();
        System.out.println("Congratulations!")
    }
}
```

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

```java
public class Stanfurd {
    public void cheer() {
        System.out.println("Stanfurd is 2cool4skool");
    }

    public void enrollStudent() {
        numStudents += 1;
        if (numStudents % 1000 = 0) {
            System.out.println("We have " + numStudents + " students!");
        }
        System.out.println("Congratulations!")
    }

    public void expelStudent() {
        students -= 1;
    }
}
```

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

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

STOP!

DON'T PROCEED UNTIL EVERYONE IN YOUR GROUP HAS FINISHED AND UNDERSTANDS ALL EXERCISES IN THIS SECTION!

## 8 HOF

Use these two interfaces for the following problems.

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

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

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

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

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.

```java
public class Add10 implements UnaryFunction {

    private static final Adder add = new Adder();

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

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).

```java
public class AddX0 implements UnaryFunction {

    private static final Add10 addTen = new Add10();
    // not necessary to init here but recommended
    private int tensPlace;

    public AddX0(int num) {
        tensPlace = num;
    }

    public int apply (int x) {
        int result = x;
        for (int i = 0; i < tensPlace; i++) {
            result = addTen.apply(result);
        }
    }
}
```

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.

```java
public class Multiplier implements BinaryFunction {

    private static final Adder add = new Adder();
    // not necessary to init here but recommended

    public int apply(int x, int y) {
        int result = 0;
        for (int i = 0; i < y; i++) {
            result = add.apply(result, x);
        }
        return result;
    }
}
```

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.

```java
public int apply(int x, int y) {
    if (y < 0) {
        return -apply(x, -y);
    } else {
        int result = 0;
        for (int i = 0; i < y; i++) {
            result = add.apply(result, x);
        }
        return result;
    }
}
```

STOP!

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