CS61BL: Data Structures & Programming Methodology Summer 2014

Midterm 1 Solutions

1 Equality Checks (6 points)

Write a `probablyEquals` method that takes in two objects and returns `true` if one or more of the following are true:

- The two objects are equal (`.equals`) to each other.
- The two objects are equal (==) to each other.
- The two objects have the same `.toString()` representation.
- Calling `.hashCode()` on both objects returns the same `int`.

Otherwise, `probablyEquals` returns `false`. Your method should never crash given *any* input. You may assume that for any object instances `x` and `y`, `x.equals(y)` will return the same value as `y.equals(x)`.

Note: `hashCode()` is a method that returns an `int`. All objects inherit this method from the `Object` class. For the purposes of this problem, `toString()` is another method that all objects inherit from the `Object` class and that returns a `String` representation of the object.

Solution:

```
public static boolean probablyEquals( Object obj1, Object obj2 ) {

    // Put this first, consider the case where both are null
    if (obj1 == obj2) {
        return true;
    }
    // Must be before method calls!
    if (obj1 == null || obj2 == null) {
        return false;
    }
    // Strings are objects, use .equals
    if (obj1.equals(obj2)) {
        return true;
    }
    if (obj1.toString().equals(obj2.toString())) {
        return true;
    }
    // ints are primitive, use ==
    if (obj1.hashCode() == obj2.hashCode()) {
        return true;
    }

}
```

Comments: The most common mistake by far was forgetting to do a null check. It's important to make sure objects aren't null before calling methods on them, otherwise you get exceptions. A try/catch is also an acceptable way to catch these errors... provided you catch the right one. You should *not* catch `IllegalArgumentException` in this problem, that is a very specific `Exception` and not what we're looking for. You needed to look for `NullPointerException`, or more generally `Exception`. There is also no reason to be throwing new exceptions, as this causes the code to crash.

We were also testing you on knowing the difference between `.equals` and `==`. For primitive values, like int, you must use ==. For Objects (like String),`==` only compares memory addresses rather than contents, so we must use `.equals` to compare Objects.

2 Whose `Line` is it Anyway? (4 points)

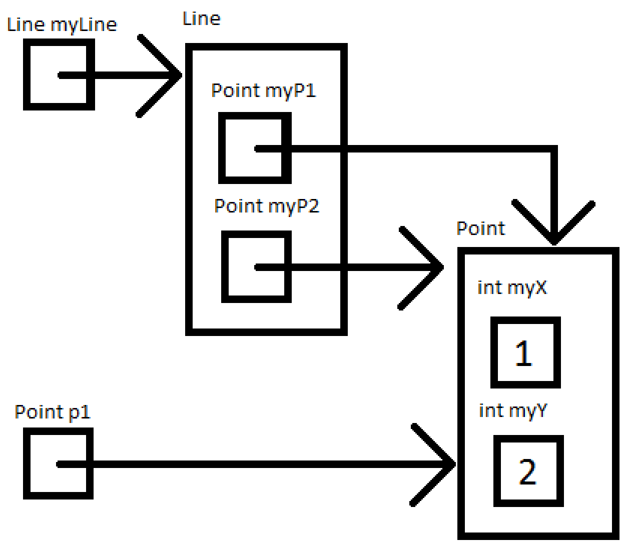
Examine the following implementations of a `Point` class and `Line` class:

```java
public class Point {
    private int myX;
    private int myY;

    public Point(int inputX, int inputY) {
        this.myX = inputX;
        this.myY = inputY;
    }

    public int getX() {
        return this.myX;
    }

    public int getY() {
        return this.myY;
    }
}
```

```java
public class Line {
    private Point myP1;
    private Point myP2;

    public Line(Point p1, Point p2) {
        this.myP1 = p1;
        this.myP2 = p2;
    }

    public String toString() {
        String toRtn = "";
        toRtn += ("(" + myP1.getX() + "," + myP1.getY() + "),");
        toRtn += ("(" + myP2.getX() + "," + myP2.getY() + ")");
        return toRtn;
    }

    public static void main(String[] args) {
        Point p1 = new Point(1, 2);
        Line myLine = new Line(p1, p1);
        // Your code here
        System.out.println(myLine);
    }
}
```

(a) In the space below, draw the resulting box-and-arrow (a.k.a. box-and-pointer) diagram after executing lines 18 and 19 of the `Line` class (up until the `// Your code here` line).

Solution:



(b) In the space below, rewrite the `// Your code here` with a single line of code so that the program prints out:

```
(1,2),(3,4)
```

Solution:

```
Sol1: myLine = new Line(p1, new Point(3,4));
Sol2: myLine.myP2 = new Point(3,4);
```

3 Building a Knapsack (12 points)

The following code represents a knapsack that can carry items and keep track of the total weight of the items it contains:

```java
import java.util.ArrayList;

public class Knapsack {
    protected ArrayList<String> itemNames;
    private ArrayList<Integer> itemWeights;
    private final int weightCapacity;
    private int totalWeight;

    public Knapsack(int weightCapacity) {
        itemNames = new ArrayList<String>();
        itemWeights = new ArrayList<Integer>();
        this.weightCapacity = weightCapacity;
        this.totalWeight = 0;
    }

    public void addItem(String name, int weight) {
        itemNames.add(name);
        itemWeights.add(new Integer(weight));
    }

    public void removeItem(String name) {
        int itemIndex = itemNames.indexOf(name);
        itemNames.remove(itemIndex);
        itemWeights.remove(itemIndex);
    }

    public int getTotalWeight() {
        return totalWeight;
    }
}
```

Example of usage (after you implement part a):

```java
Knapsack myKnapsack = new Knapsack(5);
myKnapsack.addItem("Banana", 1);
myKnapsack.addItem("Water Bottle", 3);
myKnapsack.getTotalWeight(); // should return 4
myKnapsack.removeItem("Banana");
myKnapsack.getTotalWeight(); // should return 3
```

(a) Add code to the `Knapsack` class so that after every method call, the `totalWeight` instance variable always equals the total weight of all of the items in the `Knapsack`. You may or may not have to use all of the boxes below.

Code added immediately after line **Solution:** 16, 17, or 18 :

Solution: totalWeight += weight;

Code added immediately after line **Solution:** 22 or 23 :

Solution: totalWeight -= itemWeights.get(itemIndex);

Code added immediately after line ________:

Comments: Some solutions lost points for indexing into the `ArrayList` rather than using the `get` method. Furthermore, in the `removeItem` method, the code in the second box cannot be added after line 24, because we cannot `get` the weight of an item if we've already removed that weight from `itemWeights`.

There were other ways to complete this problem (such as re-calculating the total weight after each method call by iterating through `itemWeights`), but these other solutions were generally less efficient and clean than the solution above. (These alternative solutions received full credit.)

Also note that it was not necessary to cast the items in the `itemWeights` `ArrayList` from `Integer` to `int` due Java's auto-unboxing. Those who did cast, however, were not deducted points. (Those who were deducted points attempted to cast by using methods that did not exist or otherwise incorrect syntax that caused compiler errors.)

(b) Add code to the `Knapsack` class so that `Knapsack`s will never:

- contain two items with the same name
- have a total weight greater than its weight capacity
- contain an item with negative weight
- have a negative capacity

If a user tries to modify a `Knapsack` to have any of the above properties, an `IllegalStateException` (from `java.lang`) should be thrown with an informative error message. Assume that the code from part (a) has been implemented correctly. You may or may not have to use all of the boxes below.

Code added immediately after line 16:

Solution:

```
if (itemNames.contains(name)){
    throw new IllegalArgumentException(name + " already exists.");
}
if (totalWeight + weight > weightCapacity){
    throw new IllegalArgumentException("Not enough capacity to carry"
                                        + name + ".");
}
if (weight < 0){
    throw new IllegalArgumentException("Invalid input: " + name + "
                                        has negative weight");
}
```

Comments: By checking items before they are added, we can make sure that (1) there are no duplicate items, (2) a knapsack's total weight will never exceed its weight capacity, and (3) items of negative weight will never be contained in a knapsack. It is important that these checks are done immediately *before* the invariants for a `Knapsack` object are violated, so the state of a knapsack will always be consistent even after an `IllegalArgumentException` is thrown.

Code added immediately after line 9:

Solution:

```
if (weightCapacity < 0)
    throw new IllegalArgumentException("Error: specified negative
        capacity for a knapsack.");
```

Comments: The weight capacity is initialized in the constructor, so the check should be done anywhere within its body.

(c) What is another case of error checking we should add to `Knapsack`?

Solution: There were several accepted solutions:

- check that items exist in the `Knapsack` before attempting to remove them
- check that `String` names are not `null`
- check that `String` names are not empty (e.g. "")

Comments: There were several common incorrect answers:

- check that item does not have weight of 0 (items - like feathers - can have weight of 0)
- check that `totalWeight` is not negative (this invariant was previously checked by (b))
- check that `Knapsack` is not too heavy to carry (hilarious, but not correct)

(d) Fill out the following template for a `ValueKnapsack` class that keeps track of items' values in addition to their weights. You should use inheritance effectively. Don't worry about error-checking for this part. Assume that parts (a) and (b) have been implemented correctly.

Solution:

```
public class ValueKnapsack extends Knapsack {

    private int totalValue;
    private ArrayList<Integer> itemValues;

    public ValueKnapsack(int weightCapacity) {
        super(weightCapacity);
        totalValue = 0;
        itemValues = new ArrayList<Integer>();
    }

    public void addItem(String name, int weight, int value) {
        super.addItem(name, weight);
        totalValue += value;
        itemValues.add(value);
    }

    public void removeItem(String name) {
        totalValue -= itemValues.remove(itemNames.indexOf(name));
        super.removeItem(name);
    }

    public int getTotalValue() {
        return totalValue;
    }
}
```

4 Acronym Extractor (7 points)

We want to code an `extractAcronym` method that returns the acronym of an input `String`. We will be using the `#` character in place of whitespace characters for this problem (you may assume that the input will not have any whitespace characters). An acronym consists of the first non-`#` letter of the input `String` and all non-`#` characters that immediately follow any `#` character. For example, `extractAcronym("Not#a##Number")` would return `"NaN"`. If there are `#` characters at the end of an input `String`, we ignore them.

(a) In the table below, provide test inputs for `extractAcronym` that cover at least 4 generalized input cases (not including the provided example). Include the input `String`, expected output `String`, and what your test case is testing for. Do not reuse the example. Assume that the input and return value will never be `null`.

Solution:

Input `String`	Expected output	What are you testing for?
"Not#a##Number"	"NaN"	(The example) Tests that `extractAcronym` can handle cases with multiple consecutive # characters.
"Hello#"	"H"	Tests input with trailing # characters
"#Hello"	"H"	Tests input with leading # characters
"###"	""	Tests input with only # characters
"Hello"	"H"	Tests input with no # characters
""	""	Tests empty string as input

Comments: Your solution should not:

- combine multiple test cases
  (e.g. tests input with both trailing and leading # characters)
- have redundant test cases
- make specialized cases for attributes not relevant to the specs
  (e.g. distinguish between non-# characters, repeated words, words are long, etc...)
- be too specific outside of edge cases
  (e.g. test exactly 3 consecutive # characters between single non-# characters)

You did not receive points if you didn't provide a correct test description.

(b) Implement the `extractAcronym` method:

Solution:

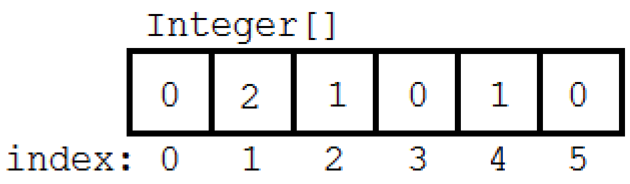
```
public static String extractAcronym(String input) {
    String result = "";
    boolean looking = true;
    for (int i = 0; i < input.length(); i++) {
        if (looking && input.charAt(i) != '#') {
            result += input.charAt(i);
            looking = false;
        }
        if (input.charAt(i) == '#') {
            looking = true;
        }
    }
    return result;
}
```

Alternate solution:

```
public static String extractAcronym(String input) {
    return input.replaceAll("((?<=[^#])[^#]*#*)|^#*", "");
}
```

5 Vote Iterator (10 points)

Write an iterator that takes in an `Integer` array of vote counts and iterates over the votes. The input array contains the number of votes each selection received. For example, if the input array contained the following:



then calls to `next()` would eventually return 1 twice (because at index 1, the input array has value 2), 2 once, and 4 once. After that, `hasNext()` would return `false`.

Provide code for the `VoteIterator` class below. Make sure your iterator adheres to standard iterator rules.

Solution:

```
import java.util.ArrayList;
import java.util.Iterator;

public class VoteIterator implements Iterator<Integer> {

    private Integer[] inputVotes;
    private ArrayList<Integer> votesArrayList;
    private int votesIndex;

    public VoteIterator(Integer[] input) {
        votesArrayList = new ArrayList<Integer>();
        for (int i = 0; i < input.length; i++) {
            for (int j = 0; j < input[i]; j++) {
                votesArrayList.add(new Integer(i));
            }
        }
        votesIndex = 0;
        inputVotes = input;
    }

    public boolean hasNext() {
        return votesIndex < votesArrayList.size();
    }
    public Integer next() {
        if (hasNext()) {
            votesIndex++;
            return votesArrayList.get(votesIndex - 1);
        } else {
            return null; // or throw some exception
        }
    }

    public void remove() {
        if (votesIndex > 0) {
            inputVotes[votesArrayList.get(votesIndex - 1)]--;
        }
    }
}
```

Comments: This solution is the simplest in terms of code. It does a lot of preprocessing in the constructor by adding all of the votes to an `ArrayList` and then returns the votes one at a time with calls to `next`. Note that you still have to store the original input array for `remove` to work properly.

Alternate solution:

```
import java.util.Iterator;
public class VoteIterator implements Iterator<Integer> {

    private Integer[] votes;
    private int voteIndex; // Which vote within same vote type
    private int bucketIndex; // Type of vote

    public VoteIterator(Integer[] input) {
        votes = input;
        voteIndex = 0;
        bucketIndex = 0;
    }

    public boolean hasNext() {
        if (voteIndex < votes[bucketIndex]) {
            return true;
        } else {
            for (int i = bucketIndex + 1; i < votes.length; i++) {
                if (votes[i] > 0) {
                    return true;
                }
            }
        }
        return false;
    }

    public Integer next() {
        if (voteIndex < votes[bucketIndex]) {
            voteIndex++;
            return bucketIndex;
        } else {
            for (int i = bucketIndex + 1; i < votes.length; i++) {
                if (votes[i] > 0) {
                    bucketIndex = i;
                    voteIndex = 1;
                    return bucketIndex;
                }
            }
        }
        return null; // or throw some exception
    }

    public void remove() {
        if (voteIndex == 0) {
            return; // or throw some exception
        } else {
            votes[bucketIndex]--;
            voteIndex--;
        }
    }
}
```

Comments: This is a more conceptually straightforward solution that iterates through the input votes array. It keeps two counters: one that indexes into the input votes array and another that keeps track of how many votes of the current vote type have already been returned.

6 Malicious Mallory (5 points)

Eve and Mallory are lab partners in CS61BL. Unfortunately for Eve, Mallory doesn't get along with anyone. One day, when Eve isn't looking, Mallory codes the following method and adds calls to it in Eve's code:

```
 1 import java.util.ArrayList;
 2
 3 public void method() {
 4     ArrayList a = new ArrayList();
 5     String msg1 = "Code not working? Try turning your computer off and on again!";
 6     String msg2 = "Is your code running? Then you better go catch it!";
 7     String msg3 = "You might have forgotten a semicolon somewhere.";
 8     a.add(new IllegalArgumentException(msg1));
 9     a.add(new ArrayIndexOutOfBoundsException(msg2));
10     a.add(new NumberFormatException(msg3));
11     int index = (int) (Math.random() * 10);
12     if (index >= 3) return; throw a.get(index);
13 }
```

This code is meant to do nothing 70% of the time, and throw one of three random Exceptions the other 30% of the time. Note: `Math.random()` returns a random `double` in the range $[0,1)$.

(a) There is an error with this code. Explain what the error is, and whether it is a compile time error, a runtime error, or a logic error.

Solution: `ArrayList` isn't given a generic type, so `a.get(index)` will give an object of static type `Object` when `throw` is expecting an object of static type `Throwable` (or any of its subclasses). This is a compile time error.

(b) Cross out one line of code above. Rewrite this line of code below so that the code compiles, runs, and does what Mallory intended.

Solution:
Cross out line 12:
`if (index >= 3) return; throw (RuntimeException) a.get(index);`

Alternatively, you could have crossed out line 4:
`ArrayList<RuntimeException> a = new ArrayList<RuntimeException>();`

Comments: You have to cast as a `RuntimeException` (unchecked exception) and not just an `Exception`. Throwing a potentially checked exception would also require the method header to say `throws Exception`.