CS61BL: Data Structures & Programming Methodology — Summer 2015

Instructor: Joseph Moghadam — **Midterm 1** — July 10, 2015

| | |
|---|---|
| **Name:** | (Dexter) Xucheng Guo |
| **Student ID Number:** | 26958386 |
| **Section Time:** | 08 |
| **TA:** | Alice |
| **Course Login:**<br>cs61bl-?? | JF |
| **Person on Left:**<br>Possibly "Aisle" or "Wall" | Aisle |
| **Person on Right:**<br>Possibly "Aisle" or "Wall" | NIKAT PATEL |

- Fill out ALL sections on this page. (1 point)
- Do NOT turn this page until the beginning of the exam is announced.
- You should not be sitting directly next to another student.
- You may not use outside resources other than your 1 page cheat sheet.
- You have 110 minutes to complete this exam.
- Your exam should contain 5 problems over 12 pages.
- This exam comprises 10% of the points on which your final grade will be based (30 points).
- If you have a question, raise your hand and a staff member will come to help you.
- Make sure to check for corrections / clarifications that will be periodically added to the screen at the front of the room.
- Best of success. Please relax – this exam is not worth having a heart failure over.

## 1 What Would Java Do? (4 points)

Imagine the following blocks of code occur in a method in some Java class. For each block of code, write down what it would print out if it were run. If you think the code won't compile, just write "Compile-time error". If you think the code will compile but create a runtime error, write what you think the runtime error would be.

Assume that the class imports `java.util.*`

(a)

```java
List<String> l = new ArrayList<String>();
l.add("macarons");
System.out.println(l.get(0));
```

What would it do? macarons :

(b)

```java
String s1 = "macaroons";
String s2 = null;
System.out.println(s1.equals(s2));
```

What would it do? false :

(c)

```java
String s1 = null;
String s2 = "break!";
System.out.println(s1.equals(s2));
```

What would it do? NullPointerException :

(d)

```java
Object[] arr = new Object[3];
System.out.println(arr[1]);
```

What would it do? null :

(e)

```java
Object[] arr = new Object[3];
Object o = arr[1];
System.out.println(o);
```

What would it do? null :

(f)

```java
int[] arr1 = {1, 2, 3};
int[] arr2 = {1, 2, 3};
System.out.println(arr1 == arr2);
```

What would it do? false ;

(g)

```java
int[] arr1 = {1, 2, 3};
int[] arr2 = {1, 2, 3};
arr2 = arr1;
System.out.println(arr1 == arr2);
```

What would it do? true ;

(h)

```java
int[] arr1 = {1, 2, 3};
int[] arr2 = arr1;
arr2[1] = 50;
System.out.println(arr1[1]);
```

What would it do? 50 ;

(i)

```java
String o1 = "wug";
String o2 = "capybara";
String o3 = "pangolin";

String[] arr1 = {o1, o2, o3};
String[] arr2 = {o1, o2, o3};
arr2[2] = "noodles";
System.out.println(arr1[2]);
```

What would it do? pangolin ;

(j) Recall the `Counter` class. It stores a single instance variable, `myValue`, an `int`. Assume it is a public variable, and it is initialized to 0 in the constructor.

```java
Counter o1 = new Counter();
Counter o2 = new Counter();
Counter o3 = new Counter();

Counter[] arr1 = {o1, o2, o3};
Counter[] arr2 = {o1, o2, o3};
Counter c = arr2[2];
c.myValue++;
System.out.println(arr1[2].myValue);
```

What would it do? 1 ;

## 2 Draw It Out (6 points)

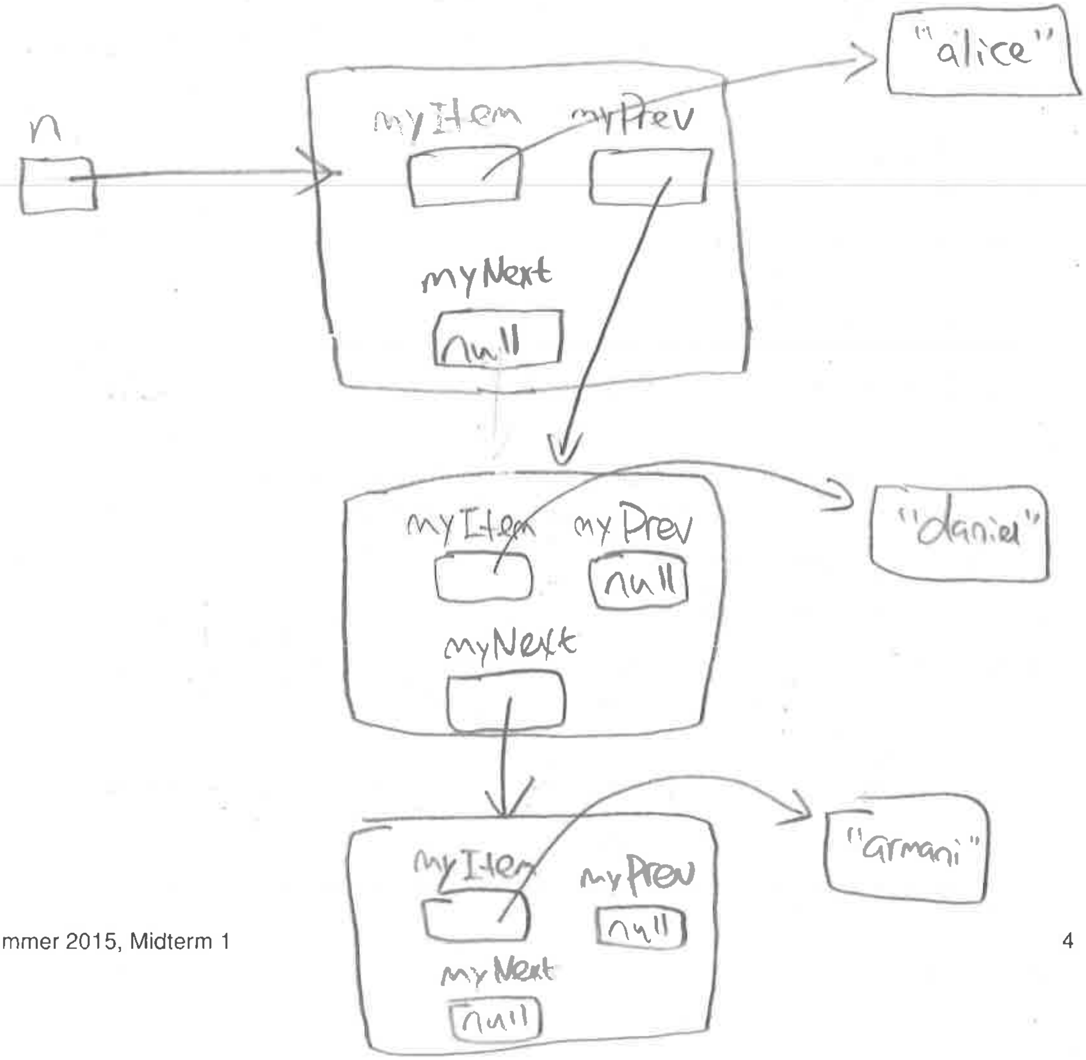
Imagine you have the following class somewhere:

```
public class ListNode {
    public Object myItem;
    public ListNode myPrev;
    public ListNode myNext;

    public ListNode(Object item, ListNode prev, ListNode next) {
        myItem = item;
        myPrevious = prev;
        myNext = next;
    }
}
```

For each of the following, draw the box-and-pointer diagram that exists after running the following code (assume nothing has been garbage collected). No need to draw stack frames. Also, no need to label the types of objects or references. However, please put names where appropriate.

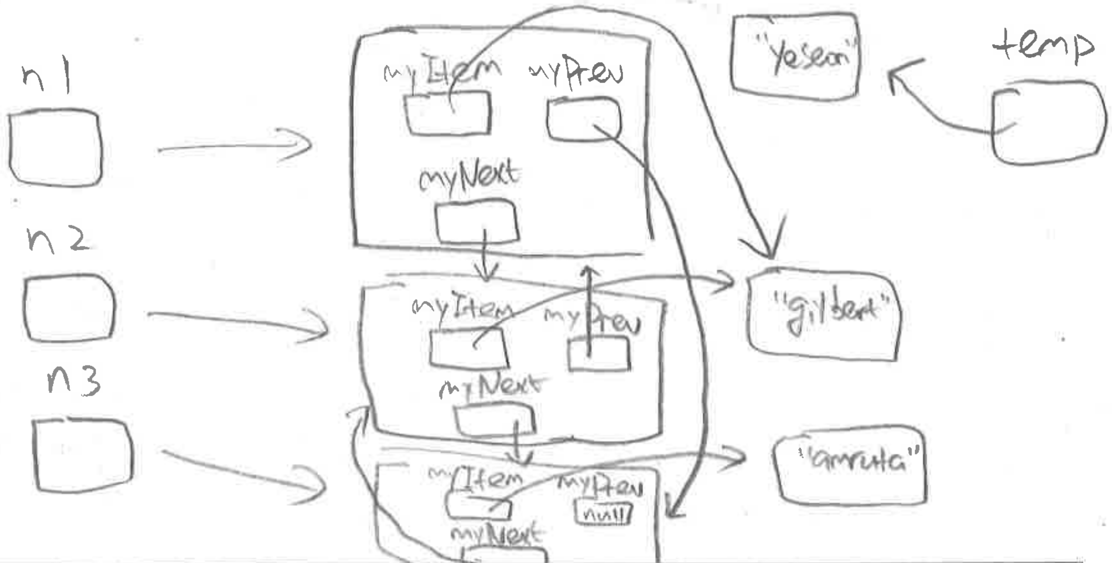
(a)

```
ListNode n = new ListNode("alice", new ListNode("daniel", null, new ListNode("
    armani", null, null)), null);
```



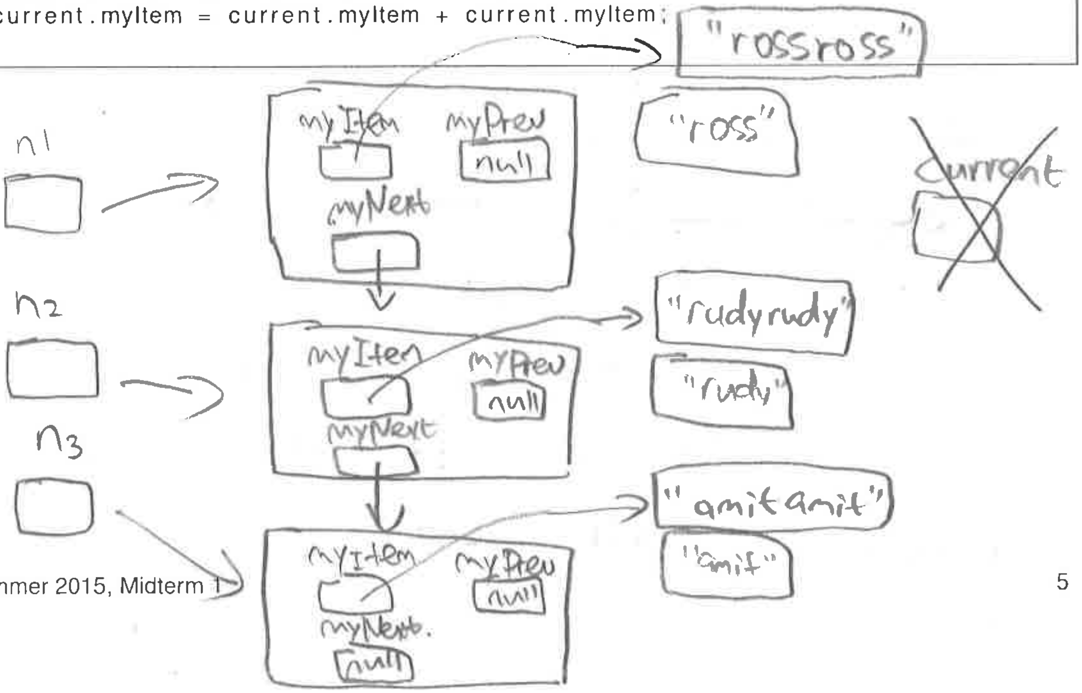
(b)

```java
ListNode n1 = new ListNode("yeseon", null, null);
ListNode n2 = new ListNode("gilbert", null, null);
ListNode n3 = new ListNode("amruta", null, null);

n1.myNext = n2;
n1.myNext.myPrev = n1;
n1.myPrev = n3;
n3.myNext = n2;
n2.myNext = n1.myPrev;
String temp = n1.myItem;
n1.myItem = n2.myItem;
```



(c)

```java
ListNode n1 = new ListNode("ross", null, null);
ListNode n2 = new ListNode("rudy", null, null);
ListNode n3 = new ListNode("amit", null, null);
n1.myNext = n2;
n2.myNext = n3;

for (ListNode current = n1; current != null; current = current.myNext) {
    current.myItem = current.myItem + current.myItem;
}
```



## 3 Relationships (6 points)

Below are listed several pairs of terms. For each pair, imagine a fellow student asked you to explain the difference between the two terms. How would you explain them to this student? For each pair, write one or two sentences (no more than two sentences will be graded). More points will be given for answers that are more accurate, precise, and that demonstrate knowledge of class concepts.

(a) Static method and non-static method

Static method belongs to the class and the memory is allocated for it only once. Non-static method belongs to the specific instances of the class.

(b) Static variable and instance variable

Static variable belongs to the class and is shared between all instances of that class. Non-static variables belongs to the instances of the class and is seperate from different instances.

(c) Static type and dynamic type

Static type is what a compiler looks for and dynamic type is what the run time looks for.

(d) Instance variable and local variable

Instance variable belongs to an instance of a class and local variable is created temperately inside one { } like inside method definition or a while loop.

(e) Abstract class and interface

Abstract class ~~ha~~ can have abstract methods and an instance of an Abstract class may not be created. Interface can only have public abstract methods and public static final variables.

(f) Overriding a method and overloading a method

Overriding a method is when you ~~exten~~ inherit from a class and ~~th~~write a method that have exact same name and argument list as that in the superclass. Overloading is when inside the same class you write a method that have the same name as one other method but different argument list.

(g) `.equals` and ==

.equals is a method inside Object that is intended to compare the content of two objects. == is an operator that compares whether the two ~~refre~~ references refer to the same object.

## 4 Running Iterator (6 points)

Imagine you have a long array of numbers like so:

```
1, 2, 3, 1, 1, 1, 1, 1, 1, 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 3
```

It would take an array of size 20 to represent these numbers. However, we claim that you can *compress* this sequence by taking advantage of the fact that it has so many elements in a row that are the same. We claim that you could represent the same array as above with the following two arrays:

```
Values:  1, 2, 3, 1, 4, 1, 3        Counts:  1, 1, 1, 6, 2, 8, 1
```

(Note: This is similar to, but not exactly the same as the sparse vector shown in lecture)

The values array stores all of the items in the array, but if a number is repeated multiple times in a row, it only stores the number once. Instead, there is a separate array, counts, which stores the number of repetitions for each corresponding item in values, so no information about the original array is lost. Notice for this example, even with two arrays, there are only 14 numbers stored in total, less than the original 20! So we have saved some space.

In a moment we'll ask you to write an iterator for this class. **Before you write the code**, first provide at least three example (uncompressed) arrays you would use to test out your code. For each, explain in one sentence why it's an important test case (i.e. what edge case is it testing?) Full points will be given for covering different edge cases that are *not already covered by our example.*

1. {}. An array with no element!

2. {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}. contains only one element.

3. {1, 8, 6, 2, 7, 9, 3, 4, 0, 5}. no repetitive elements.

Here's the framework for the class:

```
import java.util.Iterator;

public class CompressedArray implements Iterable<Integer> {
    private int[] myValues;
    private int[] myCounts;

    public CompressedArray(int[] arr) {
        // Compress the input array, storing the results in myValues and myCounts
        // Don't worry about how this works. Just assume it does.
```

```
            ...
    }

    @Override
    public Iterator<Integer> iterator(){
        return new CompressedArrayIterator();
    }

    private class CompressedArrayIterator implements Iterator<Integer>{
        private int valueIndex;
        private int currentCount;

        public CompressedArrayIterator() {
            valueIndex = 0;
            currentCount = 0;
        }

        @Override
        public boolean hasNext {
            return valueIndex < myValues.length;
        }

        @Override
        public Integer next() {
            int toReturn = myValues[valueIndex];
            currentCount++;
            if (currentCount >= myCounts[valueIndex]{
                valueIndex++;
                currentCount =0;
            }.
            return toReturn;
        }
    }
}
```

The iterator for this class ***should return items as they were in the original, uncompressed array.*** It should obey all the normal rules of iterators. In the example given above, the first call to `next` would return 1. The second would return 2. The third would return 3. The the following *six* calls to `next` would all return 1. And so on.

We've given the `Iterator` class some instance variables you can use. You shouldn't need any more. You also shouldn't need any more lines than given, though you can add more if you feel you must. If you feel you need far more lines of code than given, you're making the problem more complicated than it needs to be.

## 5 Oops! (7 points)

In lab, you worked with an `Account` class that could store a money balance. It supported methods that allowed a user to deposit money into or withdraw money from the account. It had the following methods:

```
/* Returns the current balance in this account */
public double balance() {...}

/* Adds the amount to this account's balance */
public void deposit(double amount) {...}

/* Tries to withdraw the amount from this account.
 * Returns true if succeeds.
 * If there isn't enough money, doesn't withdraw,
 * and returns false.
 */
public boolean withdraw(double amount) {...}
```

We made one small modification. We've changed the code to use `double`s rather than `int`s.

Now consider a slight twist on this concept: the `CreditCard` class. It supports the following methods and constructor:

```
/**
 *
 * @param acct
 *            A bank account used to pay off debt on this credit card.
 * @param creditLimit
 *            The maximum amount of debt this card is allowed to have.
 * @param interestRate
 *            The amount by which the debt is increased at the end of the
 *            month.
 * @throws IllegalArgumentException
 *             if acct is null, creditLimit is negative, or interestRate is
 *             negative
 */
public CreditCard(Account acct, double creditLimit, double interestRate)
    throws IllegalArgumentException {. . .}

/**
 * Purchases an item. A user can buy as much as they want, no matter what is
 * in their bank account. The cost is just added to a temporary debt on the
 * credit card. With one caveat: The amount of debt on the card cannot
 * exceed the credit limit. In this case, the user fails to buy the item.
 *
 * @param itemName
 *            Name of the item being purchased
 * @param cost
 *            The cost of the item
 * @return True if is successfully able to buy the item, false if not.
 * @throws IllegalArgumentException
 *             if cost is negative
 */
public boolean buyItem(String itemName, double cost)
    throws IllegalArgumentException {. . .}

```

```
/**
 * Reduces the debt on the credit card by this amount. Also removes the
 * money from the bank account, if the bank account has sufficient funds.
 * Else, it does not remove money from the account.
 *
 * @param amount
 *             Amount to pay off debt by
 * @return True if the bank account associated with this card has enough
 *         money to pay off the debt.
 * @throws IllegalArgumentException
 *              if amount is negative, or more than the amount of debt that
 *              exists on the card. In this case, nothing happens.
 */
 public boolean payDebt(double amount) throws IllegalArgumentException {. . .}

/**
 *
 * @return A list of the names of items bought in the current month.
 */
 public List<String> itemsBoughtThisMonth() {. . .}

/**
 * Automatically called at the end of the month. Any remaining debt on the
 * card is increased by itself * the interest rate. After this method is
 * done, the items bought in the current month should be emptied out, in
 * preparation for the new month.
 */
 public void passTime() {. . .}
}
```

Please read the descriptions of the methods carefully to figure out what they should do and how they work together. Your task is to write the complete class (please do not re-write the comments). The class does not need a main method. You don't need to write the `Account` class – assume is already exists.

```
public class CreditCard {
    Account myAcct;
    double myCreditLimit;
    double myInterestRate;
    double myDebit;
    List<String> myItemsBought;
    public CreditCard (Account acct, double creditLimit, double interestRate)
        throws IllegalArgumentException {
            if(acct == null || creditLimit < 0 || interestRate <0)
                throw new IllegalArgumentException ();
            myAcct = acct;
            myCreditLimit = creditLimit
            myInterestRate = interestRate;
            myDebit = 0;
            myItemsBought = new ArrayList<String>();
    }.
public boolean buyItem ( String itemName , double cost)
  throws IllegalArgumentException {
      if (cost < 0)
         throw new IllegalArgumentException ();
      if (myDebit + cost > ~~myCreditLimit~~)
                             myCreditLimit
         return false;
      myDebit += cost;
      myItemsBought.add (itemName);
      return true;
 }.
 public boolean payDebt (double amount) throws IllegalArgumentException {
      if (amount < 0 || amount > myDebit)
         throw new IllegalArgumentException ();
      If (~~amou~~!myAcct . withdraw (amount))
         return false;
      myDebit -= amount;
      return true;

 }
 public List<String> itemsBoughtThisMonth () {
      return myItemsBought;

 }
 public void passTime () {
      myDebit *= myInterestRate;
      myItemsBought = new ArrayList<String>();

 }

}
```