Homework 1

This homework is divided into three parts. The first set of questions test your knowledge of java. The second set tests your knowledge of lists. For these questions, you may ask for advice on how to approach them (i.e., English descriptions) but not on how to write the code.

The final set of problems focuses on problem solving, not syntax. These problems are no harder or easier in java than they are in English. What's being measured here is your ability to think flexibly. Because of this, you may not ask for advice on how to approach these problems.

All code should be in a package specific to the question number.

Java Technical Questions

These questions test your knowledge of java. None of these involve writing a data structure.

1. Baseline

Tests: All the basic java knowledge you need for the rest of the homework
Rank: Very Easy (hopefully)

Before you tackle the tough questions, let's make sure you've got the basics of java down. This class doesn't have a good story, it's just a bunch of random methods to make sure you understand specific pieces of java syntax. If you need help with any part of this question (and this question only), you may ask other people in class for help. If you get stuck on the java part of other questions, look back at how you did the java in this part.

Rather than describe what each method is supposed to do, we've given you a java file with each method defined (but not implemented). Each one has comments explaining what you should do.

---

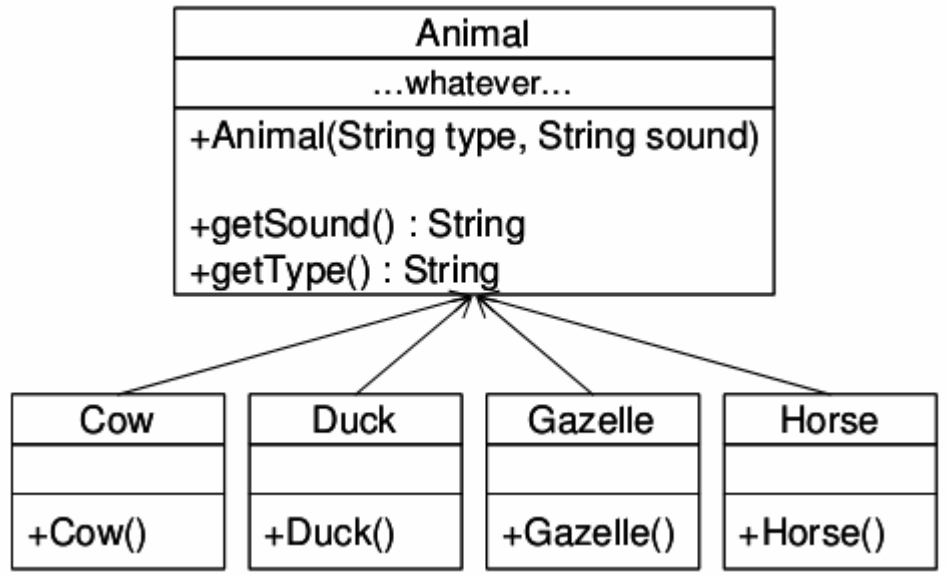
2. Old MacDonald

Tests: Inheritance
Rank: Easy

Old MacDonald has a farm, which is where Big Macs come from. Before they were Big Macs, they were animals.

The animals make different sounds.

Class	Type	Sound
Cow	Cow	moo
Duck	Duck	quack
Gazelle	gazelle	
Horse	Horse	neigh



Create the classes. Make sure that they return the exact type and sound shown above (gazelle's don't have a voice box so their sound is an empty string). Case matters. Animal (also known as the parent class or super class) does not have any public properties (also no protected or package private). You may make private ones.

The subclasses have no properties and no methods other than a constructor that takes no arguments. The subclass doesn't need arguments such as type and sound because cows already know sound a cow makes. For this class to work, it must inherit from the super class Animal and call the super class' constructor (if you don't remember how, your text book, the slides and Google can help).

---

3. Fix the Code

Tests: Code fixing and debugging skills
Rank: Medium

The TAs wrote a program. Unfortunately, one of the TAs is really, really bad at programming. It has both compilation errors and logic errors. Also, their style is pretty bad – terrible variable names, no comments, duplicate code, all sorts of problems. i didn't have time to fix their code so i want you to do it. The code is in the project you downloaded. Good luck.

List Technical Questions

For these questions, we have you build on your labs. You've already written MyArrayList and MyLinkedList, now we want you to add a few more methods.

4. Enhanced MyArrayList

Tests: Array List
Rank: Medium

You made an array list in lab. Now you're going to add a few new methods to it (yes, you can use your lab version as the base for this one). Your new array list should have this interface:

MyArrayList<T>
you figure this part out *no public properties*
+MyArrayList() : ctor +MyArrayList(int initialCapacity) : ctor +MyArrayList(List<T> items) : ctor +add(T item) : void +addAll(List<T> items) : void +clear() : void +clone() : MyArrayList<T> +contains(T item) : boolean +get(int index) : T +getSize() : int +insert(int index, T item) : void +remove(int index) : void +removeRange(int start, int end) : void +update(int index, T item) : void

`MyArrayList()`	Null constructor. Allows you to create the list without doing anything.
`MyArrayList(` `  int initialCapacity)`	Create a list with the specified initial capacity.
`MyArrayList(List<T> items)`	Create a list and add a bunch of items to it. You might want to check how many items you're adding before allocating memory in order to avoid resizes.
`add(T item)`	Add an item. Items added to a list go at the end.
`addAll(List<T> items)`	Add a bunch of items
`clear()`	Delete everything in the list.
`clone()`	Create a copy of the list. Specifically, create a new list containing copies of each item. This is sometimes called a shallow copy because the lists are different but the items in them are the same (i.e., point to the same chunk of memory).
`contains(T item)`	Return true if the list contains the item. Remember that you cannot compare the data in an object using ==.
`get(int index)`	Return the item at the specified index.

`getSize()`	size() is a stupid method, method names should be verbs, so we're going to call this getSize().
`insert(int index, T item)`	Insert an item at the specified location. If a list has five items (0-4) and you insert at 2, item 2 is the new item and the old items 3 and 4 become items 4 and 5.
`remove(int index)`	Remove the item at the specified index. If a list has five items (0-4) and you remove item 2, items 3 and 4 become items 2 and 3.
`removeRange(int start, int end)`	Remove the items at the specified positions. If a list has five items (0-4) and start=1, end=3, items 1, 2 and 3 are removed and item 4 becomes item 1.
`update(int index, T item)`	Change the data at the specified index to the given data.

---

5. Enhanced MyLinkedList

Tests: Linked List
Rank: Medium

Now let's make a linked list with the exact same interface (minus one constructor):

`MyLinkedList<T>`
you figure this part out *no public properties*

`+MyLinkedList() : ctor`
`+MyLinkedList(List<T> items) : ctor`
`+add(T item) : void`
`+addAll(List<T> items) : void`
`+clear() : void`
`+clone() : MyLinkedList<T>`
`+contains(T item) : boolean`
`+get(int index) : T`
`+getSize() : int`
`+insert(int index, T item) : void`
`+remove(int index) : void`
`+removeRange(int start, int end) : void`
`+update(int index, T item) : void`

See question 4 for explanations of what the methods do.

---

6. Bejeweled 1D

Tests: Manipulate pointers in a Linked List

Rank: Medium High

You probably know the game Bejeweled. If you match 3 gems in a row, they blow up and you get points. It's normally played with a 2D grid of jewels but we're going to start simple and just do a single row. Here's how it will work: given a linked list of jewels

1. Position = 0
2. Scan the line to see if there are three gems of the same color together. Don't worry about if there are 4 in a row – if there are, just act like it was only 3.
3. If there are, remove them then re-scan the line again, from the beginning (position 0), continuing until there are no more sets of 3 matching gems left. Reset the position to 0.
4. If there are not, rotate the three gems at Position (pos, pos+1, pos+2) clockwise – given the nodes `A->B->C`, rotate them to `B->C->A`.
5. Re-scan the line.
6. If no jewels are removed, increment Position.
7. Go back to step 2
8. When the end of the line is reached, return to step 1.

We will implement three play methods: `playOneMove()`, `playOnce()` and `play()`. `playOneMove()` makes a single rotation and however many scans are needed after that to remove matching gems. `playOnce()` starts at position 0 and plays until it reaches the end of a line. `play()` plays until the end of game conditions are met. The game ends when there is no jewel type that has three or more jewels anywhere on the board.

The game looks like this:

Bejeweled1D
+ isOver() : boolean; + play(BejeweledMap map) : int + playOnce(BejeweledMap map) : int + playOneMove(BejeweledMap map, int position) : int

The class BejeweledMap is just a linked list that takes Strings.

BejeweledMap
+ head : LinkNode<Character>
+ BejeweledMap(String map) + isEmpty() : boolean + remove(int index) : void + size() : int + toString() : String + worthPlaying() : boolean

The following is a sample trace of `playOnce()`:

```
Step 0: AABBCBCCADDD
```

```
  scan: AABBCBCCADDD                  Score=1
Step 1: AABBCBCCA -> ABABCBCCA
  scan: -
Step 2: ABABCBCCA -> AABBCBCCA
  scan: -
Step 3: AABBCBCCA -> AABCBBCCA
  scan: -
Step 4: AABCBBCCA -> AABBBCCCA      Score=2
  scan: AACCCA                      Score=3
  scan: AAA                         Score=4
  scan: game over
```

Notes:

- Maps will not start with any chains (3 consecutive gems of the same color).
- Remove chains before checking whether to rotate.
- `playOneMove()` checks for chains, rotates then checks once more for chains.
- Before playing, make sure it's worth it. Check that there are at least 3 gems of one color.

Problem Solving Questions

For these questions, the java isn't the hard part. You need to figure out (preferably on paper first, in English) how to solve these problems. If you can figure out a good psuedocode approach, the java should be fairly simple.

How hard or easy a question is depends in part on whether you're using the best data structure. For questions 8 and 9 you are allowed to use java's java.util.HashMap class.

You may ask questions about java syntax (i.e., how do i loop through a dictionary, how do i get a character from a string, etc.) but not about the approach to take to solve the problem. Since that's what we're measuring here, you have to do that part on your own.

7. Dictionary – a List implementation

Tests: Problem solving
Rank: Very Easy

We've discussed Dictionaries in class. They let you store and retrieve data using a key. We've also discussed Lists, which allow you to store one type of data. Your job is to make a dictionary work **using only a single List**. If you think it will help, you may inherit from or use your `MyArrayList` or `MyLinkedList`.

`MyDictionary<K,V>`
No public properties

Up to you to design private properties

`+add(K key, V value) : void`
`+contains(K key) : boolean`
`+get(K key) : V`
`+size() : int`

---

8. Pomposity Measure

Tests: Problem solving
Rank: Easy

A large vocabulary is a sign of intelligence. Or maybe it's a sign of pompous. Whatever the situation, let's create a way to measure how big a person's words are.

`PersonalityCalculator`
No public properties Up to you to design private properties

`+calculatePomposity(List<String> words) : double`
`+calculatePomposity(List<String> words,`
`  List<String> noiseWords) : double`
`+getMostPompous(Map<String, List<String>> people) : String`
`+getMostPompous(Map<String, List<String>> people,`
`  List<String> noiseWords) : String`

Here's what everything does:

`calculatePomposity(`
`  List<String> words,`
`  List<String> noiseWords)`
`: double`
	Calculate the average length of the given words. Example: The words "I", "ate", "a", "pickle" have a total length of 11 and an average length of 2.75. You can get the length of a word with the String's `length()` method.
	
	A noise word is a word we don't care about. Common noise words include "a", "and" and "the". If you are given any noiseWords (noiseWords != null), do not include any word found in that list in your calculation. In the above example, "a" is a noise word and is therefore not included, so the average length is 10 / 3 = 3.33. To find out if a word is in the noise list, use the list's `contains()` method.

`calculatePomposity(`
`  List<String> words)`
`: double`
	Same as above but without any noise words (i.e., don't de-noise). If you write more than one line of code for this, you're doing it wrong.

`+getMostPompous(`
`  Map<String, List<String>>`
`  people,`
`  List<String> noiseWords)`
`: String`
	You are given a dictionary where the key is a person's name and the value is a list of words that person uses. Your job is to figure out which of those people has the highest pomposity score and return their name. Every person uses the same set of noise words.

`+getMostPompous(`
`  Map<String, List<String>>`
`  people)`
`: String`
	Same as the above but without any noise words. Again, if you write more than one line of code, you're doing it wrong.

For those interested, the field of Information Retrieval has done a lot of research on analyzing documents (called a corpus) for interesting information. They use techniques such as de-noising, stemming (converting words such as *get*, *gets* and *getting* to just *get*) and frequency baselines to do things like determining what a document is about. It's also a big part of how search engines work.

---

9. Anagram Checker

Tests: Problem solving
Rank: Medium

Given two words, determine whether they are anagrams. There are many ways to do this but if you use the right data structure (and use it the right way) this problem is pretty easy.

`AnagramChecker`
i can't imagine you'd need any properties

`+areAnagrams(String word1, String word2) : boolean`
`+getAnagrams(String word, List<String> candidates) :`
`  List<String>`

In case you don't remember the (very brief) lecture on string methods, you might find the `charAt(index)` method useful. Example:

```
char firstLetter = word.charAt(0);
```

As mentioned in class, every primitive data type has an object wrapper class and the one for char is `Character`. We've also discussed auto-boxing and unboxing in class and on the forums. The important part is this – you can't store a `char` in a list (map, tree, etc.) but you can store a

`Character`, and while you have to define the list (etc.) to use type `Character`, you don't actually need objects of that type, you can just use `char`. Example:

```
List<Character> alphabet = new ArrayList<>();
char letterA = 'a';
alphabet.add(letterA);
char firstLetter = alphabet.get(0);
```

---

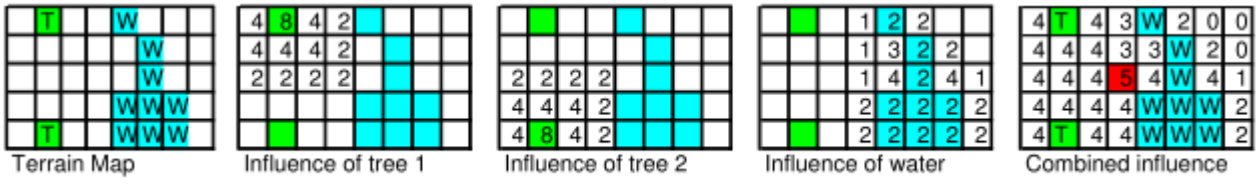
10. Picnic Planner

Tests: Multi-dimensional arrays, array manipulation, bounds-testing
Rank: Medium High

For this problem, you're going to use an advanced spatial analysis artificial intelligence technique known as an influence map to determine which position on a map best achieves some goal. In this case, that goal is to go on a picnic.

Naruto wants to take Sasuke on a picnic. Sasuke loves trees and water and especially loves cherry blossoms. He is nervous around people and therefore a spot with some privacy is best.

In an influence map, you place values on a map and propagate their "influence" (how much they affect the areas around them) outwards using a decaying propagation function, which is a fancy sounding phrase for "sitting right next to the lake is better than sitting a block from it". In this problem, trees are worth 8, have a propagation distance of 2 and a decay function of halving, which means that the spot where the tree is is worth 8, the eight squares that are right next to the tree are 4 (1 tile away, half the value of the tree) and the squares next to those are worth 2 (2 tiles away, half the value of the neighboring cells). Squares after that are worth nothing – they're too far away to really get the benefit of being next to a tree.



Example Influence Map (8x5, 2 trees, 1 river made up of 9 water tiles)

In an influence map, each object has influence on the area around it. The influences of each object are added together on a new map called a desirability map. For example, a spot near one tree is nice and worth 4 points but a spot near two trees is nicer and worth 8. Once all the object's influences are added together, the spot with the highest score is the most desirable position. In the example above, the position in the center of the map (marked in red) is the best spot for a picnic since it is near the water (+1) and in view of two trees (+2 each).

An important point – having a picnic near a lake is nice. Having one in a lake is not. The desirability map should only include values for blank spots – we want to have a picnic near a tree but not in it.

Feature	Value at Center	Propagation Distance	Decay Function
Tree	8	2	cut by half
Water	2	1	cut by half
Cherry Tree	16	3	cut by half
People	-8	2	cut by half

The input map is a two-dimensional array of characters. Valid values are `T`, `W`, `C`, `P` and `' '` (space). The output is either the position of the best spot on the map or the desirability map used to determine that position. We use ints for the desirability map because none of the values will ever be fractional (and because they're easier for you to print while debugging) but in the real world these would probably be floats.

MapAnalyzer
No public properties *Up to you to design private properties*
`+getDesirabilityMap(char[][] terrainMap) : int[][]` `+getBestPosition(char[][] terrainMap) : GridPoint`

GridPoint
`+x : int` `+y : int`

The `[][]` in `char[][]` refers to the x (column) and y (row) dimensions. In computer science, unlike in math, (0,0) is the top left corner of the grid, not the bottom left.

Don't worry about ties. All of our test cases will always result in exactly one "best" option.

Make sure you write helper methods. Don't write one giant `getDesirabilityMap()` method, write a small method that calls other methods, each of which does one thing and does it well. One method a lot of students find helpful to write is often called

```
clamp(int x, int max) : int,
```

which takes a number and forces it to fall within the bounds of 0 and max. i'll let you figure out where that might be helpful (you'll learn pretty quickly if you use the right test cases).

You hopefully already know this but you **absolutely want** to write methods that print off the maps so that you can debug them. If you don't have a method called something like `printMap(map)`, you are a super-genius or a masochist.
