HW1 CodeCamp

Assignment by Nick Parlante

For this first homework, you will run through a series of small coding problems to get warmed up with Java code, Collection classes, and unit tests. Get in the habit of pulling up the javadoc pages for String, Collection, and leveraging the built-in libraries. Download the hw1CodeCamp.zip file from the course page. That contains a hw1CodeCamp directory that is an Eclipse project -- import it into Eclipse to get started. This homework is due at midnight at the end of Thursday January 13.

You have a variety of small coding problems to solve, each with unit tests. You should fill out the set of unit tests for each problem until you are confident that your solution is correct. The deliverable is that there are 5 or more assertEquals() or other assertions for each public method. The starter code has some tests filled in for some of the problems to get you started, and you should add more tests. Part of this assignment is about experimenting with unit tests to really nail down the correctness of your code beyond the usual "it appears to work" level. Since the problems are only moderately hard, have well-defined input/output interfaces, and you can write unit tests, it is a reasonable goal that you will turn in literally perfect code for all the problems -- zero bugs. Producing a body of zero bug code does not happen by accident. It requires some organized effort at testing.

Our grading will be pretty simple -- we just have automated tests for each problem that verify that your solution works correctly for a variety of inputs. The grading in that case will mostly just be about correctness. That said you should still try to write clean code and use decomposition.

CS108 homework ground rules: the starter code will often include some boilerplate such as the prototypes for methods you need to write. A starter method may include a throwaway line like "return 0;" just so it compiles when you first load the project. Your code can assume that all inputs are formatted and structured correctly. Your solution should work correctly when presented with valid inputs, and we will not worry about invalid inputs (yet). If your code takes in a parameter such as a String or a List, you may assume that the String or List passed in is not null, unless null is specifically mentioned in the specification. In other words, null is not a valid String or List. The empty string and empty list are valid of course, but in Java those are different from null. Your code should never change the public interfaces given for homework problems. Very often, we have tests that call your code in various ways, and obviously that only works if your code keeps the same interface as given in the starter code. For the same reason, you should leave your classes in the default package, so our testing code can find it. You are free to **add** additional or helper methods -- adding extra methods will not confuse our testing code. Very often, my own solution decomposes out private helper methods for parts of the problem, pulling some complexity out of the main methods.

String Code

String blowup(String str)

Returns a version of the original string as follows: each digit 0-9 that appears in the original string is replaced by that many occurrences of the character to the right of the digit. So the string "a3tx2z" yields "attttxzzz", and "12x" yields "2xxx". A digit not followed by a character (i.e. at the end of the string) is replaced by nothing.

int maxRun(String str)

Given a string, returns the length of the largest run in the string. A "run" is a series of zero or more adjacent characters that are the same. So the max run of "xxyyyz" is 3, and the max run of "xyz" is 1.

boolean stringIntersect(String a, String b, int len)

Given 2 strings, consider all the substrings within them of length len. Returns true if there are any such substrings which appear in both strings. Compute this in O(n) time using a HashSet. Len will be 1 or more.

CharGrid

The CharGrid class encapsulates a 2-d char array with a couple operations.

int charArea(char ch)

Given a char to look for, find the smallest rectangle that contains all the occurrences of that char and return the rectangle's area. If there is only a single occurrence of the char, then the rectangle to enclose it is 1x1 and the area is 1. If the character does not appear, return an area of 0. For example, given the grid…

```
abcd
a cb
xbca
```

The area for 'a' is 12 (3 x 4) while for 'c' it is 3 (3 x 1). The second row contains a space char, but that's still just a regular char.

For testing, you can set up a 2-d char[row][col] array literal like this (row 0 is "cax")

```
char[][] grid = new char[][] {
        {'c', 'a', 'x'},
        {'b', ' ', 'b'},
        {' ', ' ', 'a'}
    };
```

int countPlus()

Look for a '+' pattern in the grid made with repetitions of a character. A + is made of single character in the middle and four "arms" extending out up, down, left, and right. The arms start with the middle char and extend until the first different character or grid edge. To count as a +, all the arms should have two or more chars and should all be the same length. For example, the grid below contains exactly 2 +'s...

```
  p
  p    x
ppppp xxx
  p  y x
  p yyy
zzzzzyzzz
  xx y
```

Hint: consider decomposing out a private helper method to avoid repeating the code to measure each of the four arms.

TetrisGrid

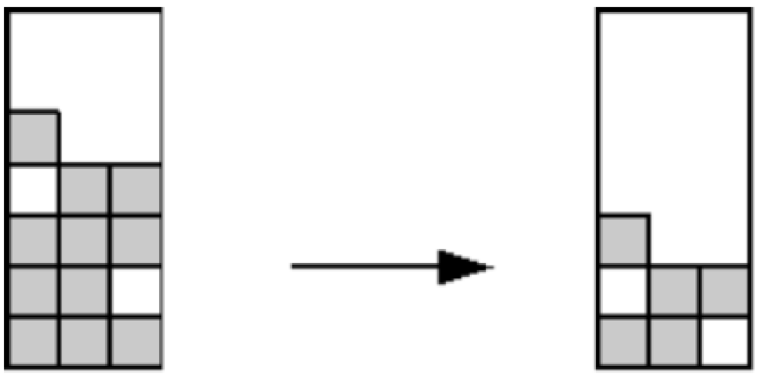
The TetrisGrid class encapsulates the classic rectangular board for the game Tetris (we'll play around with Tetris on homework 2, so here's a chance to get a head start on that code). We'll store the Tetris board as a grid of booleans, where true is a filled square, and false is an empty square. We'll use the convention that grid[x][y] refers to a cell in the board, with grid[0][0] representing the lower left square in the board, x growing to the right, y growing up (the standard Cartesian coordinate system). In the Tetris code, grid[x][y] is a natural way to think about the game, but notice that it's different from a grid[row][col] convention.

Constructor -- the TetrisGrid constructor should take in a boolean[][] grid argument. The width and height of the grid will both be at least one. For example, below is a grid that is width 2 and height 3. The 2-d array literal syntax is row/col oriented, so our grid[x][y] appears rotated 90 degrees clockwise.

```
boolean[][] grid = new boolean[][] {
  {false, false, true},    // this is grid[x=0][...]
  {true,  true,  false}    // this is grid[x=1][...]
};
```

void clearRows()

The one key method in TetrisGrid is clearRows() which should delete the full rows in the grid, shifting the rows above down and adding empty rows at the top, like this:



There is a simple getGrid() that exposes the grid stored in TetrisGrid, so unit tests can call clearRows() and then getGrid() to check the resulting grid.

Collections

<T> int sameCount(Collection<T> a, Collection<T> b)

In the Appearances class, the static Appearances.sameCount() method takes in two collections -- A and B -- containing a generic <T> element type. Assume that the T elements implement equals() and hashCode() correctly, and so may be compared and hashed. The elements are in no particular order. Every element in A appears in A one or more times, using .equals() to compare elements for equality (the standard definition of equals() for java collections). Likewise every element in B appears one or more times. sameCount() counts the number of elements that appear in both collections the same number of times.

For example, with the collections {"a", "b", "a", "b", "c"} and {"c", "a", "a", "d", "b", "b", "b"} it returns 2, since the "a" and the "c" appear the same number of times in both collections. Use hashing to compute the number of appearances efficiently.

class Taboo<T>

Most of the previous problems have been about single methods, but Taboo is a class. The Taboo class encapsulates a "rules" list such as {"a", "c", "a", "b"}. The rules define what objects should not follow other objects. In this case "c" should not follow "a", "a" should not follow "c", and "b" should not follow "a". The objects in the rules may be any type, but will not be null.

The Taboo noFollow(elem) method returns the set of elements which should not follow the given element according to the rules. So with the rules {"a", "c", "a", "b"} the noFollow("a") returns the Set {"c", "b"}. NoFollow() with an element not constrained in the rules, e.g. noFollow("x") returns the empty set (the utility method Collections.emptySet() returns a read-only empty set for convenience).

The reduce(List<T>) operation takes in a list, iterates over the list from start to end, and modifies the list by deleting the second element of any adjacent elements during the iteration that violate the rules. So for example, with the above rules, the collection {"a", "c", "b", "x", "c", "a"} is reduced to {"a", "x", "c"}. The elements in bold -- {"a", **"c", "b",** "x", "c", **"a"**} -- are deleted during the iteration since they violate a rule.

The Taboo<T> class works on a generic <T> type which can be any type of object, and assume that the object implements equals() and hashCode() correctly (such as String or Integer). In the Taboo constructor, build some data structure to store the rules so that the methods can operate efficiently -- note that the rules data for the Taboo are given in the constructor and then never change.

A rules list may have nulls in it as spacers, such as {"a", "b", null, "c", "d"} -- "b" cannot follow "a", and "d" cannot follow "c". The null allows the rules to avoid making a claim about "c" and "b".