Program #2 Design Notes

Away we go. Start early… to win!

Part 1 - Tracker console

Remember - this is function over form. In the console, we just want to prove that our tracker works for our Shakespeare files.

Tokens - Use my `Program2Helper.parseLine()` to turn a line from a file into a list of tokens. It's on the k: drive, program2 folder.

Sort - You can sort an `ArrayList` using the method `Collections.Sort()`. I made `WordCount` is-a `Comparable`, so that it will sort based on the count. We will cover this in Ch 16 Search & Sort.

Reverse - Oops. Your list is sorted in ascending order. Use `Collections.Reverse()` to fix this. EZ.

WordTracker - When you're building your tracker, `countWords()` for the file should call `countWords()` for a line, which should call `countWord()` for each word in the line. Yes? Case - I store my initial word with its case intact, like "Hamlet". But comparisons are case-insensitive. The `String` class has a nice method for this: `equalsIgnoreCase()`.

2 mains - I actually have two classes (though this is definitely optional):

- `Program2` - `main()` for the gui
- `Program2Console` - `main()` for the console

Early results - Here are some early results from my implementation of Part 1 on our test files.

File test1.txt

- Tracker: num diff words=9, total num words=10
- My word list=[test [2], system [1], broadcast [1], emergency [1], the [1], of [1], a [1], is [1], This [1]]

File test161.txt

- Tracker: num diff words=10, total num words=30
- My word list=[ProfBill [7], WilliamW [5], AllyA [5], MattM [4], SteveS [3], DwayneD [2], that [1], How's [1], Ally [1], ConnorC [1]]

File macbeth.txt

- Tracker: num diff words=4102, total num words=20371
- My Top 10 words are: [The [764], and [602], to [460], of [427], I [344], a [288], you [269], that [245], in [225], is [212]]
- My Top 10 words at least 4 chars long are: [which [82], Enter [81], shall [68], Macbeth [67], their [62], Rosse [49], would [48], should [46], there [43], Where [38]]

Let me know (email me) if your results match or are different.

Some createWordList() pseudo-code

One of our more complex (and important) methods in P2 is `createWordList()` in your `WordTracker`. It's this guy:

```
ArrayList<WordCount> createWordList( int topN, int minWordSize,
                          ArrayList<String> excludedWords);
```

Here's so pseudo-code of my solution:

sort existing list of WordCount objects in descending order
[hint: Collections.sort() and Collections.reverse()]

create a new, empty list of WordCount objects to return
count = 0 // used to count words as they are add to the new list
for each WordCount in Tracker list
    valid = true
    if word is too small, then valid = false
    if word is in excluded list, then valid = false;

    if valid, then {
        add word to new list
        count++
        if count >= topN then break loop
    }
}
return new list

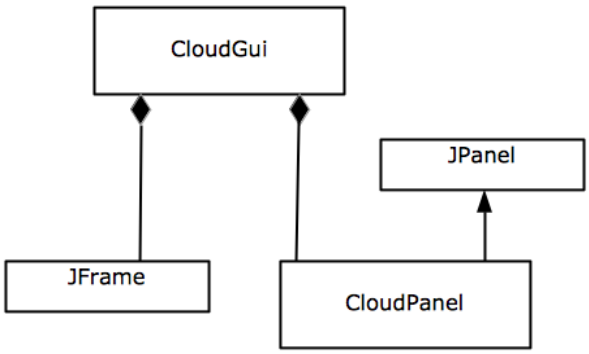
And, you can simplify your code by creating some small private methods to test words:

- `private boolean isWordTooSmall( String s, int minSize) { … }`
- `private boolean isWordExcluded( String s, ArrayList<String> excludedList) { … }`

Part 2 - Cloud GUI

We know:

- We'll be writing to a `JPanel`. We'll create a subclass of panel, like Program #1, where most of the work happens. Here's one possible UML diagram. If you like `CloudGui` is-a `JFrame` better (like our text does), then that works too.



- Use `drawString()` in the `Graphics` class to draw text on a `JPanel`.

- We've used `setColor()` to set the color for future drawing. There's also `setFont()` for future text drawing.

Let's get started on some basic research. Open issues include:

- How can we save the `JPanel` drawings to an image file (jpg or png)?
  I got this one. It's savePanelToImageFile() in Program2Helper.

- Can we draw text that is vertical? (like the Hamlet example in the assignment)

- Of course, the big salami is placement of each word. It seems pseudo-random. Random, but clumped. Any ideas? (Your first try should definitely just place words at random. Get that running and then start tweaking!)

On `drawString()` in the `Graphics` class...

- There's good stuff and examples on page 882 and 892 of our text.
- The coordinates (x,y) are the lower left corner of the string you are drawing. This is different from our circles in Lab01, where (x,y) was the upper left of the shape.
- I have added `getTextBounds()` to `Program2Helper`. This method returns a rectangle that is the bounding box for the string, given a `Font` and `Graphics` object. With this `Rectangle`, you can place the string so that it doesn't draw off the edge of the panel.

- Also, I refer you to a method that we will use later, I think. Each `Rectangle` has a method: `boolean intersects( Rectangle r)`. This method returns true if two `Rectangle` objects intersect. This may help keeping our strings from overwriting each other.

Program2Helper.java

Get the new Program2Helper.java on the k: drive. So, there are currently 4 methods:

- `public static ArrayList<String> parseLine(String line)` - parses a line from a file into tokens

- `public static Rectangle getTextBounds(String text, Font f, Graphics g)` - returns the "bounding box" of a string given its font and panel Graphics object

- `public static void savePanelToImageFile(JPanel panel, String fileName)` - save the drawings on a JPanel to a PNG file

- `public static void createTest161Tracker( WordTracker wt)` - create a test WordTracker if you are struggling with Part 1.

The CloudString interface

I have also added these two Java files:

- ❖ `CloudString.java` - the `CloudString` interface
- ❖ `CloudStringAbs.java` - an abstract class that implements `CloudString`. I added this class to pass along some delicate (cough) code to help you place your `CloudString` objects.

`CloudString` is just a more complicated version of our FunCircle objects from Lab01.
Future stuff:

- ★ My first instinct was to define the color, font, and location of my `CloudString` objects in my gui ctor. But that won't work because you need a Graphics object to determine the bounding box (rectangle) of your `CloudString`. So, you must place your `CloudString` objects in the `paintComponent()` method. Remember: only place them once otherwise, they'll change every time the panel is redrawn.

- ★ I have a Cloud/Refresh menu. I create a new CloudPanel using the latest, greatest settings the user has specified in my other menus.

```
// words is my WordCount list created using the tracker
this.panel = new CloudPanel( words);
this.frame.setContentPane( this.panel);
this.frame.pack();
```

CloudPanel.paintComponent(g) pseudo-code

Things can get complicated in paintComponent(). Divide and conquer... to win!

```
void paintComponent( Graphics g) {
  super.paintComponent(g)   // always!

  if( firstTime) {    // only create & place CloudString objects once!
    createAllCloudStrings( g)   // need the Graphics to place CloudString
  }

  for each CloudString {
    draw it
  }
}
```

Continue on…

```
private void createAllCloudStrings( Graphics g) {
  create a new list to hold the CloudString objects we create
  for each WordCount {
    cs = new CloudString for the WordCloud
    chooseColor( cs)
    chooseFont( cs)
    chooseLocation( cs, g)
  }
}
```

See that? Place each step in fancying up your CloudString is a separate method. This allows your to start small (one font size) and grow (set font size based on word count).

The location can be done in baby steps:

- Place randomly on the panel
- Place randomly on the panel, without going off the edge
- Place randomly on the panel, without overlapping any other CloudString objects
- Place randomly on the panel, and then clump the CloudString with others

Clumping

If you can randomly place your words, without going off the edge of the panel and without overlapping, then you're **golden**. The next step is clumping. Clumping means moving your CloudString objects closer together… clumping together.

After randomly placing your CloudString… here's my pseudo-code for clump():

```
clump( CloudString cs) {
  mid = middle point in the panel
  p = point of cs location
  forever {
    x = one pixel closer to middle
    y = one pixel closer to middle
    p2 = new point( x, y)
    set location of cs to p2

    check cs for overlaps with all other CloudString objects
    if( overlap) {
      set location of cs back to p
      break loop      // reached an overlap, so we're done
    }
    if( p2 == middle) {
      break the loop      // reached the middle, so we're done
    }
  }
}
```

`JPanel` has `getWidth()` and `getHeight()` methods.

I made checking for overlap its own private method. Like this:

```
boolean checkForOverlap( cs) {
  overlap = false
  for each CloudString object cs2 {
    if( cs2 != cs) {       // don't check ourselves!
      if (cs intersects cs2) {
        overlap = true
        break loop
      }
    }
  }
  return overlap
}
```