CS201 Discussion 4

Debugger, JohnSort
Using the Debugger

- Ambient help page on the Eclipse Debugger
- Eclipse documentation on Debug View
- Debugging Advice
Using the Debugger
Using the Debugger

• Step Over
• Step Into
• Step Return
• Resume
• Terminate
The Debugger In Action
Your turn!

• A student with poor coding style has come to you for debugging help with their CirclesCountry code (snarfable). The code is almost working, but needs slight fixes in order to be perfectly correct.

• See if you can use the debugger to fix the student’s solution. Place breakpoints at important places in the code, watch the variables change at each step, and try to make fixes based on what you observe.

• (Since this is an APT you’ve done before, you could probably do this without the debugger, but still try to practice using it as much as possible)
The clue to what’s wrong in your code is in the values of your variables.

1. **Set breakpoints at key locations** in your code.
   - Wherever the problem clearly manifests itself
   - Wherever the key objects are constructed or key relationships are being established.

2. **Look at code critically**: Examine all the variables that are visible at the breakpoint

3. **Step slowly through your code**, checking all your variables.
   - Use Step Over and Step Return only when you are sure.

4. **Iterate** and add more breakpoints when you see something amiss. Use the Resume button to quickly advance to the next breakpoint.
Problem Statement

As some of you may know, there is no name better than JOHN. Let's define the rules for comparing names. Each letter has a weight (A = 1, B = 2, ..., Z = 26). The weight of a name is the sum of the weights of all its letters. For example, the name MARK has weight 13 + 1 + 18 + 11 = 43.

When comparing two names, the one with the larger weight is considered better. In case of a tie, the one that comes earlier lexicographically is better. But there is one exception - the name JOHN is the best name of all.

You are given a String[] names, each element of which contains a single name. Sort the names from best to worst and return the sorted String[].

```java
class TheBestName {
    public String[] sort(String[] names) {
        // fill in code here
    }
}
```
Solving sorting APTs

• For the APTs in this set, you can solve them using this general structure (obviously, for some you’ll need a few extra steps)

  1. Write an inner class which uses Comparator to sort the input

  2. Call Arrays.sort() on the input and your Comparator

  3. Return the now-sorted input as the output

• This turns sorting a list into writing a method that decides which of two elements comes first – a much simpler problem!
Write an inner class which uses Comparator to sort the input

```java
public class TheBestName {
    public class Sorter implements Comparator<String> {
        public int compare(String a, String b) {
            // your code here
        }
    }

    public String[] sort(String[] names) {
        // fill in code here
    }
}
```
Skeleton for JohnSort

Call Arrays.sort() on the input and your Comparator. Return the now-sorted input as the output

```java
public class TheBestName {
    public class Sorter implements Comparator<String> {
        public int compare(String a, String b) {
            //your code here
        }
    }

    public String[] sort(String[] names) {
        Arrays.sort(names, new Sorter());
        return names;
    }
}
```
Writing compare

Recall – for the Comparator, compare(a, b) should return a negative value if a comes before b, and positive if b comes before a.

So, all that’s left to do is write the compare method to sort according to JohnSort’s specifications.
Breaking down compare methods

• Often when you write compare methods, you’ll have multiple criteria for determining (e.g. in the case of JohnSort, we have both the criteria of whether or not the name is JOHN, and the criteria of weight)

• When you have multiple criteria, drawing a decision tree before trying to write the code can be helpful.

• On the next slide we show you a simple example, but for more complicated compare methods (like some in the APT set) figuring out the order of decisions will be very helpful.
If we want to sort strings by length, and then alphabetically for same-length words, we could use the following decision tree (and translate it to the code on the right)

```java
public int compare(String a, String b) {
    if (a.length() != b.length()) {
        return a.compareTo(b);
    }
    return a.length() - b.length();
}
```
Finish JohnSort

• Using the skeleton we gave you, finish the compare method to compare two strings as the APT specifies, and complete JohnSort!