Recursion and Debugging

Purpose:
The purpose of this lab is to get some experience with the debugger and the mechanics of
recursion by looking at some simple recursive methods.

The debugging tool, \textit{jdb}:

A \textit{debugging tool}, or \textit{debugger}, is a utility that allow you run a program “under control.”
Specifically, it allows you to set \textit{breakpoints} in the program – points in the program where
execution will pause. You can then examine the status of the paused computation before resuming
it.

Most debuggers have very extensive features, and most program development systems provide a
 graphical user interface to the debugger. Here we use a basic, line-oriented debugging tool \textit{jdb},
and consider only a few fundamental features.

There are two basic commands for setting breakpoints. One causes a break to occur when a
method is entered, the other causes a break when a specified line is reached:

\begin{verbatim}
stop in class.method  -- set a breakpoint in a method
stop at class:line    -- set a breakpoint at a line
\end{verbatim}

For example,

\begin{verbatim}
stop at dandyString.DandyString.largest
stop at dandyString.DandyString:64
\end{verbatim}

To remove a breakpoint:

\begin{verbatim}
clear class:line      -- clear a breakpoint
\end{verbatim}

To display source code:

\begin{verbatim}
list                   -- display source code around current
list line number
list method
\end{verbatim}

To examine a variable or object:

\begin{verbatim}
print item            -- display object or variable
dump object          -- display all object information
locals                -- display local variables
\end{verbatim}

To perform the next statement or statements after a break:

\begin{verbatim}
next                  -- execute current line (step OVER calls)
step                  -- execute current line (step INTO calls)
step up               -- execute until the current method
                        -- returns to its caller
cont                  -- continue execution from breakpoint
\end{verbatim}

To start execution:
run -- start execution of a loaded Java class

And a few others:

!! -- repeat last command
help (or ?) -- list commands
exit (or quit) -- exit debugger

Set up:

Create a directory named Lab28/dandyString in your Java directory. Copy all the files from ~labCourse/Labs/Lab28/dandyString to your dandyString directory.

The class DandyString:

Instances of the class DandyString are just Strings with a different set of features.

(This set of features is not particularly useful, and we are not suggesting that this is the best way to implement these features. The point is simply to get some experience with recursion.)

Specifically:

```java
public DandyString (String s)
Create a new DandyString from the specified String.

public int length ()
The length of this DandyString.

public char nth (int n)
The n-th character of this DandyString.
require:
  1 <= n && n <= this.length()

public char largest ()
The largest character (in char order) in this DandyString.
require:
  this.length() != 0

public String toString ()
This DandyString as a String.
```

Note that, like String instances, instances of the class DandyString are immutable. That is, once created, they cannot be modified. The class provides no commands.

A DandyString instance has a single component, a String s:

```java
private String s;
```

With the exception of toString, each of the queries is implemented by a call to a simple, tail-recursive “helper” method, used to illustrate recursive solutions to simple problems. The helper method has the same functionality as the method it helps, but takes a String as argument. For instance, the method length is implemented as follows:

```java
public int length () {
```
where

```java
/**
 * Length of s
 */
private int length (String s) { ... }
```

is a recursive “helper” method.

Finally, here is the collection of auxiliary methods that are used to build the helper methods. These are:

```java
private boolean empty (String s)
    The specified String is empty.

private char head (String s)
    The first character of the specified String.
    require:
    !empty(s)

private String tail (String s)
    A new String that is equal to the specified String with its first character removed.
    require:
    !empty(s)

private String middle (String s)
    A new String that is equal to the specified String with its first and last characters removed.
    require:
    !empty(s)
```

To make sure that you understand the specification, give the result of each of the following expressions, given that `s` is the String "hello".

```java
head(s)
tail(s)
head(tail(s))
tail(tail(s))
head(tail(tail(s)))
tail(tail(tail(s)))
middle(s)
middle(middle(s))
```
Open the file \textit{DandyString.java} in the editor, and look at the recursive implementation of the "helper" methods. Note the use of the methods \textit{head} and \textit{tail}. (Don’t worry about the implementation of the auxiliary methods.) The local variables in the methods \textit{length} and \textit{largest} are initialized for the benefit of the debugger exercise. Such initialization is not necessary, and is not particularly good style.

**The class \textit{DandyTest}:**

This class is simply a test driver for \textit{DandyString}. Since we are going to use the debugger, and the debugger likes to grab standard input, input/output will be done in a window. Open \textit{DandyTest.java}. Note the creation of the \textit{BasicIOWindow}, and how the input and output streams are obtained from the \textit{BasicIOWindow}. (Warning: the implementation of \textit{BasicIOWindow} is fragile; its exact behavior is dependent of the Java version, operating system, and windowing environment you are running under. Do not be surprised if you get warnings when creating a \textit{BasicIOWindow}. Sigh.)

**Exercise the debugger with \textit{DandyString}:**

Compile \textit{DandyString} and \textit{DandyTest} using the “-g” option. E.g.,

```
javac -g DandyString.java
javac -g DandyTest.java
```

Any classes you want to debug must be compiled with the “-g” option.

Run \textit{Lab28.dandyString.DandyTest} just to see how it will work. A window, similar to a terminal window, will appear on the screen. Application input and output are done in this window. You are asked to key in a string, and the length of the string you enter is displayed. Terminate the program by closing the window.

Now run \textit{DandyTest} with the debugger:

```
jdb Lab28.dandyString.DandyTest
```

The dialog is dependent on the version you are using, but the debugger should respond something like this:

```
Initializing jdb...
>
```

Now we set a breakpoint at the entry to the helper method \textit{length}. We can either specify the method or the line number. That is, we can key either of the following:

```
stop in Lab28.dandyString.DandyString.length(java.lang.String)
stop at Lab28.dandyString.DandyString:59
```

(Since \textit{length} is overloaded, the argument type must be specified.)

To start execution, key:
The debugger responds with something like

```plaintext
run Lab28.dandyString.DandyTest
> VM Started:
```

(Depending on the environment, you may get an exception and a prompt

```plaintext
main[1]
```

Respond to this prompt with

```plaintext
cont
```

which instructs the debugger to continue execution.)

A window will appear with a message “Key in a string.” Key a short three or four character word in the window, and press Return or Enter

The debugger will inform you that the breakpoint has been reached,

```plaintext
Breakpoint hit: ...
```

and follow with the prompt

```plaintext
main[1]
```

At this point, key in the command

```plaintext
list
```

and note that we are at line 59, at the start of the helper method length. Let’s set a breakpoint at the end of this method as well. Type

```plaintext
stop at Lab28.dandyString.DandyString:64
```

(Cut and paste are handy!) We can now see when we enter and leave the method.

Type

```plaintext
print s
```

and you will see the value of the argument s. Key

```plaintext
locals
```

to see the parameters and currently declared local variables. Type

```plaintext
dump this
```

and you will get all the available information about the DandyString instance.

To execute the line, key

```plaintext
next
```

We are now about to execute line 60. If we key the command locals again, we will see that the variable len has been defined and has value 0.
Line 60 involves invoking the method `empty`. If we give the command `next`, we will execute the entire line. However, if we key `step`, we will “step into” the `empty` method. Key `step` and notice that we are now at line 109, the first and only line of the `empty` method. Type `list` again to see where you are.

Now key `cont`.

This tells the debugger to resume execution until a breakpoint is reached.

The next breakpoint we reach is again line 59. This is a result of the recursive call to `length` at line 63. (Key list if you don’t recall.) If we look at the parameter now, we should see that it is one character shorter: the argument is the tail of the original value.

Continue execution, examining the parameter at each breakpoint. Ultimately, you will reach an invocation for which the argument is the empty string `""`. Step through this execution by keying `next` three times. Note that no recursive call is made: the local variable `len` is assigned 0, and returned. Typing `next` one more time puts us back at the recursive call.

Now type `cont` again, and examine local variables. Repeat this, and notice how the recursion is “unwound.” Each method invocation returns to its caller.

Now edit `DandyTest` to test the method `largest`. Repeat the above process creating a script and testing `largest` with the input string “xyz”.

- For each invocation of this recursive method, display the values of `head`, `tailLargest`, and `result`.

**Post-lab:**

Submit the following, as directed by your lab instructor:

- values of expressions involving `head`, `tail`, and `middle` given above;
- assuming a `DandyString` created with argument "123", “hand traces” showing recursive method invocations resulting from the following calls:
  ```
  length()
  nth(3)
  largest()
  ```
- script produced testing the method `largest`. 