Virtual Lecture Notes (Part 2)

Recursion starts to grow on you after a while. A recursive method calls itself (which is interesting) and it loops without using a **`for`** or **`while`** statement (which is neat). So, let's return to the **`reverseInput()`** method and examine it in more detail.

```
< 1>       public void reverseInput()
< 2>       {
< 3>           System.out.print("Enter a word ('q' to quit): ");
< 4>           String aWord = in.next( );
< 5>           if(aWord.equals("q"))
< 6>              System.out.println( );
< 7>           else
< 8>              reverseInput( );
< 9>          System.out.println(aWord);
<10>       }
```

To new programmers, recursion is often disconcerting because they cannot shake the nagging feeling that the code is too sparse to accomplish the task. That feeling will not go away until the following questions are satisfactorily answered:

1. What is the flow of control through a recursive method?
2. What happens when a method calls itself?
3. How can all input values be retained, using only one simple variable (i.e., **`aWord`**)?
4. How can all of the words be printed with only the last **`println()`** method?

Each of these questions can now be answered, finally revealing the mystery of recursion.

1. What is the flow of control through the program?

< 1> When the **`reverseInput()`** method is first called, the header is the entry point to the method.
< 2> Curly brace marking the beginning of the method (matches up with Line <10>).
< 3> The user is prompted to enter a word or the letter "q" to quit.
< 4> User input is accepted and assigned to a String variable (aWord).
< 5> The value of aWord is evaluated to see if it equals the letter "q." If the condition is true, control switches to Line <6>. If the condition is false, control switches to Line <8>.
< 6> When the condition on Line <5> is true, Line <6> is executed causing a blank line to be printed and control switches to Line <9>.
< 7> Indicates the alternative branch for the condition statement when Line <5> evaluates to false.
< 8> When Line <5> is false, the method calls itself and returns to Line <1>. **This is the recursive call.**
< 9> Prints the value of the String variable (aWord). This line is only reached if the condition on Line <5> is false.
<10> Curly brace marking the end of the method (matches up with Line <2>).

As you can see, the desk check indicates that the flow of control through the method is straightforward. The key point is that if the condition on Line <5> is false, control moves to Line <8>, which calls the method again on Line <1>. As long as the condition is false, the method keeps calling itself. The only way to break out of this recursive loop is for the condition in Line <5> to become true. That is the essence of recursion.

What happens when a method calls itself?

This is where things get interesting. When a method calls itself, it essentially puts aside the problem it was working on and starts working on another task of the same kind, which is usually simpler (in this case, it is the same task but with a different word). The unfinished problems that are put aside basically accumulate in a stack.

For example, imagine that you are about to serve dinner, then realize someone forgot to wash the dishes. You cannot serve dinner until all the dishes are washed. You wash a plate, dry it, and start a stack. There are more plates to wash, so you grab another one, wash it, dry it, and stack it. You keep repeating this process until there is nothing left in the sink to wash and you have a stack of clean dishes. Now you can begin serving. So, you take the dish on the top (which was the last dish washed), put the food on it and serve it to a guest. Then you take the top dish on the stack (which was the second to last to be washed), put food on it, and serve it to another guest. Eventually, you get down to the last dish in the stack (which was the first one washed), put food on it for yourself, and sit down to eat with everyone.

When a method calls itself and cannot complete the task, the computer stacks the ~~dishes~~ unfinished methods. Once it comes to a task that can be completed, it goes back and deals with the stack of unfinished methods, one at a time, starting with the one on top of the stack. Once the stack of unfinished methods is processed, the method terminates.

How can all input values be retained using only one simple variable (i.e., aWord)?

There is only one **`String`** variable in this method, **`aWord`**; however, the computer keeps track of as many words as you want to enter. Under normal circumstances this is not possible, because when a new value is assigned to a variable, the original contents are overwritten and no longer available in memory. Yet, somehow, a single variable in a recursive method seems to be able to retain all of the values assigned to it. This is possible because each time the **`reverseInput()`** method is called, a new local variable **`aWord`** is created. Like the stack of dishes, imagine a stack of **`aWord`** variables. Each variable contains the word entered by the user each time the method is executed.

How can all of the words be printed with only the last println() method?

The computer must remember to complete all the pending calls to a recursive method. The **`println()`** statement is unfinished business, so it also goes on a stack. Recall that the following words had originally been entered:

```
Enter a word ('q' to quit): Reverse        < - - first call
Enter a word ('q' to quit): this           < - - second call
Enter a word ('q' to quit): input          < - - third call
Enter a word ('q' to quit): q              < - - termination
```

In order to accommodate each of the unfinished methods, the pending **`println()`** statements are stacked up. The stack is built one statement at a time from bottom to top each time the **`reverseInput()`** method calls itself.

Before the user enters the letter "q," the stack can be represented as shown here. The statement for the first method is on the bottom of the stack and the last unfinished **`println()`** statement goes on the top of the stack.

`System.out.println("input");`
`System.out.println("this");`
`System.out.println("Reverse");`

When the letter "q" is entered by the user and the condition in Line <5> evaluates to false, the computer takes care of any unfinished business. The println() statements are taken off the stack and executed in reverse order, top to bottom. So, whatever is placed on the stack last is executed first. Consequently, the result of the **`reverseInput()`** method is to print the words in the reverse order of the way they were entered.

Is Your Aversion to Recursion Overcome?

Recursion is one of the trickiest topics in computer science. Once you start relating it to your own personal experiences (everyone has washed and stacked dishes), trace through enough examples, and write some programs of your own, you will be convinced that recursion is just a different way of approaching a problem and nothing to fear.