CSC 102 Lecture Notes Week 9
More on Recursion
Introduction to Java GUI Library

I. **Announcements**

A. Labs 15 through 18 are out.

B. The lab quiz is on ***wednesday*** of week 10.

C. A final exam review is on ***friday*** of week 10.

D. The final exam is on ***wednesday*** of finals week:

1. 7-10PM June 12
2. Room 26-104

II. **Quiz and Exam Review Details**

A. ***Quiz:***

1. You'll do a paper design and coding during Wednesday lecture hour.
2. Then you'll move to the lab to compile, test, handin the program.

B. ***Final Exam Review:***

1. There will be a review of the final exam during Friday lecture, discussing the topics and kind of questions that will be asked.
2. There will be **NO** practice final as discussed previously in class.

III. **Definitions of Functional and Structural Recursion**

A. *Functional* is when a method calls itself, directly or indirectly.

1. The preceding examples in these notes are *functional* recursion.
2. As we saw, this can be more or less useful in Java on the practical level.
3. Lab 15 covers this topic further

B. *Structural* is when a class refers to itself, directly or indirectly.

1. A good example of structural recursion is the Node in a linked list.
2. Labs 16 and 17 cover this topic further.

IV. **Helper Methods in Recursive Solutions**

A. The idea of helper methods is common in functional recursion.

B. Helpers are typically used with recursive methods that have array parameters, such as the recursive sum example below.

C. Helper methods can also be used to make a recursive solution easier to implement, as in the palindrome method discussed in Section 13.2 of the book, and the mergesort algorithm discussed in Section 14.4.

D. Here's simple recursive summing method, illustrating the use of a helper method

```
/****
 *
 * This class illustrates how to compute the sum of an array recursively.  The
 * public sum method takes an array of integers and returns the sum of all its
 * elements.  A private "helper" method takes an array and an integer postion
 * in the array.
 *
 * The reason for the helper method is to avoid inefficient array copying to
 * create a sub-array.  Rather than creating a sub-array by copying, the helper
 * method takes a full array plus an integer position that indicates the
 * beginning of the sub-array.
```

```
 *
 */
public class RecursiveSum {

    /**
     * Return the sum of the given array.  Return 0 for an empty array.  Assume
     * the array is not null.
     */
      public int sum(int a[]) {
        return sum(a, 0);
     }

    /**
     * Return the sum of the given array, starting at the given position.  If
     * the position is equal to the length of the array, return 0.
     */
      private int sum(int a[], int position) {

        /**
         * Base Case: Return a sum of 0 if position is at the end of the array.
         */
         if (position == a.length)
            return 0;

        /**
         * Recursive Step: Return the sum of the first element of the array
         * with the recursive sum of the rest of the array.  The first element
         * is at a[position].  The rest of the array is represented by the full
         * array with the position incremented by 1.
         */
         return a[position] + sum(a, position + 1);
     }

   }
```

E. The code is in 102/examples/RecursiveSum.java and 102/examples/RecursiveSumTest.java

F. There's an interesting alternative solution here 102/examples/RecursiveSumAlternative.java and here 102/examples/RecursiveSumAlternativeTest.java

V. Recursive Solution to the "Classic" Fibonacci Sequence

A. The first two in the sequence are 0 and 1.

B. The following numbers are the sum of previous two

C. Here's a recursive solution from the book examples in 102/examples/book/ch13/fib

```
import java.util.Scanner;

/****
 *  This program computes Fibonacci numbers using a recursive method.
 */
public class RecursiveFib {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      for (int i = 1; i <= n; i++) {
```

```
            long f = fib(i);
            System.out.println("fib(" + i + ") = " + f);
         }
      }

      /**
       * Return the nth Fibonacci number.
       */
      public static long fib(int n) {
         if (n <= 2) { return 1; }
         else return fib(n - 1) + fib(n - 2);
      }

   }
```

D. We’ll discuss this method during lecture in class.

E. Here’s a version of recursive fib that outputs trace information as it runs; you can compile and execute it to see what’s going on:

```
import java.util.Scanner;

/****
 * This program prints trace messages that show how often the
 * recursive method for computing Fibonacci numbers calls itself.
 */
public class RecursiveFibTracer {
   public static void main(String[] args) {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter n: ");
      int n = in.nextInt();

      long f = fib(n);

      System.out.println("fib(" + n + ") = " + f);
   }

   /**
    * Return the nth Fibonacci number.  Output method trace information during
    * execution.
    */
   public static long fib(int n) {
      System.out.println("Entering fib: n = " + n);
      long f;
      if (n <= 2) { f = 1; }
      else { f = fib(n - 1) + fib(n - 2); }
      System.out.println("Exiting fib: n = " + n + " return value = " + f);
      return f;
   }
}
```

VI. **Recursive Palindrome Solution**

A. A palindrome is a string that is read the same forward or backward.

B. Here’s a recursive solution from the book examples in 102/examples/book/ch13/palindrome

C. Section 13.2 of the book discusses an alternate version of recursive palindrome that uses a helper method.

VII. **An Efficient and Elegant form of Sort**

A. It's called ***mergesort***.

B. The basic algorithm is this:

1. Divide the array in half.
2. Recursively merge sort each half.
3. Merge the two sorted halves.

C. Here's a solution from the book examples in 102/examples/book/ch14/mergesort

```
/****
 * This class sorts an array, using the merge sort algorithm.
 */
public class MergeSorter {
   private int[] a;

   /**
    * Constructs a merge sorter.
    */
   public MergeSorter(int[] anArray) {
      a = anArray;
   }

   /**
    * Sort the array managed by this merge sorter.
    */
   public void sort() {
      if (a.length <= 1) return;
      int[] first = new int[a.length / 2];
      int[] second = new int[a.length - first.length];
      // Copy the first half of a into first, the second half into second
      for (int i = 0; i < first.length; i++) { first[i] = a[i]; }
      for (int i = 0; i < second.length; i++) {
         second[i] = a[first.length + i];
      }
      MergeSorter firstSorter = new MergeSorter(first);
      MergeSorter secondSorter = new MergeSorter(second);
      firstSorter.sort();
      secondSorter.sort();
      merge(first, second);
   }

   /**
    * Merges two sorted arrays into the array managed by this merge sorter.
    */
   private void merge(int[] first, int[] second) {
      int iFirst = 0; // Next element to consider in the first array
      int iSecond = 0; // Next element to consider in the second array
      int j = 0; // Next open position in a

      // As long as neither iFirst nor iSecond is past the end, move
      // the smaller element into a
      while (iFirst < first.length && iSecond < second.length) {
         if (first[iFirst] < second[iSecond]) {
            a[j] = first[iFirst];
            iFirst++;
         }
         else {
            a[j] = second[iSecond];
```

```
            iSecond++;
         }
         j++;
      }

      // Note that only one of the two loops below copies entries
      // Copy any remaining entries of the first array
      while (iFirst < first.length) {
         a[j] = first[iFirst];
         iFirst++; j++;
      }
      // Copy any remaining entries of the second half
      while (iSecond < second.length) {
         a[j] = second[iSecond];
         iSecond++; j++;
      }
   }
}
```

Items discussed on Friday Week 9 to be added here.