Section 1 - Recursive Functions

9B.1.1 Recursion

We will introduce a new functional concept: ***recursion***.

Look at this example:

```
public class Sample
{
   // -------  main --------------
   public static void main(String[] args)
   {
      System.out.println( myPow(2.4, 4) + " vs. "
         + Math.pow(2.4, 4));
      System.out.println( myPow(1.23, 9) + " vs. "
         + Math.pow(1.23, 9));
      System.out.println( myPow(5535, -29) + " vs. "
         + Math.pow(5535, -29));
   }

   // -------  My Power --------------
   public static double myPow(double base, long exp)
   {
      if (exp == 0)
         return 1.;
      if (exp < 0)
         return 1. / myPow( base, -exp);
      else
         return base * myPow( base, exp-1 );
   }
}
```

As you see, I am defining my own home-rolled version of the **Math.pow()** method which I call **myPow()**. In my version, I only care about integer exponents, so I make the exponent parameter a **long**. Then, I test to see if I get the same results as **Math.pow()**, and, of course, I do. The odd thing about this method is that it calls itself!

Recursion is sometimes defined as a method calling itself, and **myPow()** certainly does that. There is more to it than that, but that usually gets the right neurons firing when thinking about the concept.

This method **myPow()** used to compute a ***base*** to an ***exp*** power, say $\mathbf{2.4^4}$ or "2.4 raised to the 4th power" which is:

```
2.4 * 2.4 * 2.4 * 2.4
```

The way this method works is by breaking the situation into cases. Ignore the **exp<0** case for now.

- **If exp == 0**, then the answer is 1, because ***ANYTHING TO THE ZERO POWER IS 1***. If you didn't know that before, now you do.
- **If exp is > 0**, Let's say **4**, then **myPow(base, 4)** ends up using the final **else** statement of the method, which will evaluate to:

```
base * myPow(base, 3);
```

So the ***function calls itself*** using the same base, but a power of 3, instead of 4. Whatever answer the function gets on that "inner" call, it multiplies by the ***base***, **4**, and that is the answer we want. This should make sense to you, since

base^4 is nothing more than **base * base^3**

In fact, this always works: **myPow(7, 5)** is computed by finding **myPow(7, 4)** and multiplying that answer by one more **7**. You don't have to follow the logic all the way down, just realize that if the function keeps calling itself, as it will do, the **exp** gets smaller and smaller until it finally hits 0, at which time, the function returns a 1. Then the function calls which have built up, start "paying off" or "unwinding" resulting in that 1 being multiplied by a **7**, that answer being multiplied by another **7**, etc. 5 times. It sounds messy, but look at the program. It's really clean and simple.

- **The exp < 0** case is for negative exponents. A negative exponent is just the ***reciprocal*** of the positive exponent (remember that high school arithmetic?). That is why that case looks as it does.

In general, a *recursive function* has *at least* two cases.

- One case returns a value without the function calling itself. There is no recursive call in this case. This is called the ***escape case.***
- The other case(s) results in the function calling itself, but the inner function call uses *different arguments* than it was passed. The new arguments that it passes to "itself" must bring us closer to the so-called ***escape case*** (above).

In our example, the escape case is when exp==0. And the other cases always result in a recursive call that brings us closer to that escape case because, for example, if we enter the method with exp equal to7, then we call myPow() recursively passing in 6 to the exp, thus bringing it closer to the ***escape case*** of 0.

As you can see, the above program works:

```
33.1776 vs. 33.1776
6.44386 vs. 6.44386
2.81572e-109 vs. 2.81572e-109
```

Section 2 - Binary Searching

9B.2.1 Need for Efficiency

There is one problem with the ***linear search***: it is not efficient. If we have an array of 1,000,000 students and we are searching for one, we'll have to compare, on average, 500,000 names with the ***key*** before we return with the position of the student. We *may* have to test all 1,000,000 (or maybe we get lucky and find it on the first test).

A ***binary search*** promises to find a student in one million names using at most 20 name/key comparisons. There's one catch: *we have to **sort** the array first.* This is not as bad as it sounds since we often store the array in sorted order anyway.

The concept is simple. We start out by comparing the ***key*** to the ***middle*** element of the (sorted) array. If the ***key*** is *greater than* the ***middle*** student, we know that ***key***, if present in the array, is in the ***upper half of the array***. On the other hand, if the ***key*** is *less than* the ***middle*** student, then the ***key***, if present, is in the ***lower half sub-array***. Either way, we take the appropriate (upper or lower) sub-array as the new array (it's now half as big as the original, thankfully) and repeat the test recursively. This gets repeated until we either find the key or the sub-array shrinks to nothing.

9B.2.2 A Binary Algorithm

Here is an example. First, we will simplify the **Student** and **StudentArrayUtilities** classes so that they contain/use only ***last names***. Now, imagine the array contains 100 elements and we are looking for the ***key*** "chloe". Assume that "chloe" is in the array. We sort the array and assume that "chloe" ends up at index 41 after the sort.

Informally (meaning, in the syntax below, wherein we sloppily identify the name "chloe" with the object for which we are searching) we are going to inspect the expression:

```
"chloe".compareTo(array[k].lastName)
```

Since the array is sorted, we know that if k < 41 the comparison will return a positive number because "chloe" is greater than **lastName**. If **k > 41**, the result will be negative because "chloe" will be less than **lastName**. If we compare "chloe" to item **41** exactly, **compareTo()** will return a **0**, indicating a perfect match. If we get a perfect match we return the ***index*** (41) to the ***client***.

We now start our binary search.

Note: You do not have to memorize the following sequence. It is done automatically by the recursion. You only need to understand the principles.

- We enter the method with the entire index range 0 to 99.
- **middleIndex** = (0 + 99)/2 = 49 (we use int arithmetic)
- "chloe" is less than the name in location 49, so we now search only the lower portion of the array, 0 to 48.
- **middleIndex** = (0+48)/2 = 24
- "chloe" is greater than the name in **middleIndex**, 24, so we search the sub-array 25 to 48.
- **middleIndex** = (25+48)/2 = 36
- "chloe" is greater than the name in **middleIndex**, 36, so we search the sub-array 37 to 48.
- **middleIndex** = (37+48)/2 = 42
- "chloe" is less than the name in **middleIndex**, 42, so we search the sub-array 37 to 41.
- **middleIndex** = (37+41)/2 = 39
- "chloe" is greater than the name in **middleIndex**, 39, so we search the sub-array 40 to 41.
- middleIndex = (40+41)/2 = 40
- "chloe" is greater than the name in **middleIndex**, 40, so we search the sub-array 41 to 41.
- "chloe" is equal to the name in **middleIndex**, 41, so we return 41

Now the methods start "unwinding," returning from the last call back to the first, passing back the 41 as they make their way back up the recursive chain. This took us 7 comparisons compared to 41 if we had gone through the list linearly.

For any name key that we want to find in the array, we go through this process until we either find the student or the size of the sub-array we are searching collapses into nothing. In an array of 1 million students, this happens in, at most, 20 comparisons.

Here is the complete binary search algorithm as implemented in a static class method of **StudentArrayUtilities**:

```
public static int binarySearch(Student[] array, String keyLast,
    int firstIndex, int lastIndex)
 {
    int middleIndex, result;

    if (firstIndex > lastIndex)
       return -1;

    middleIndex = (firstIndex + lastIndex) / 2;
    result = keyLast.compareToIgnoreCase(
       array[middleIndex].getLastName());

    if (result==0)
       return middleIndex;   //found him!
    else if (result < 0)
       return binarySearch( array, keyLast, firstIndex,
          middleIndex-1);
    else
       return binarySearch( array, keyLast,
          middleIndex+1, lastIndex);
```

```
    }
```

Notice that we always call **binarySearch()** with a smaller array than we came in with, guaranteeing that we will either find the name, or that the first and last index will eventually cross, resulting in the escape condition which returns a -1.

The complete listing is in the next section.

Section 3 - Complete Binary Search Program

9B.3.1 Student Search in Binary Form

Here is the complete listing, using the updated **Student** and **StudentArrayUtilities** classes that make use of only the ***last name***:

```
import javax.swing.*;

public class Foothill
{
   public static void main (String[] args)
   {
      Student[] myClass = { new Student("smith", 95),
         new Student("bauer",123),
         new Student("jacobs", 195),
         new Student("renquist", 148),
         new Student("3ackson", 108),
         new Student("perry", 225),
         new Student("loceff", 44),
         new Student("stollings", 452),
         new Student("charters", 295),
         new Student("cassar", 321),
         };

      StudentArrayUtilities.arraySort(myClass);
      StudentArrayUtilities.printArray("Sorted Array to be Searched:", myClass);

      String last;
      int found;

      last = "stollings";
      found = StudentArrayUtilities.binarySearch(myClass, last, 0,
            myClass.length - 1);
      if ( found >= 0 )
         System.out.println(last + " IS in list at position " + found);
      else
         System.out.println(last + " is NOT in list.");

      last = "Jacobs";
      found = StudentArrayUtilities.binarySearch(myClass, last, 0,
            myClass.length - 1);
     if ( found >= 0 )
         System.out.println(last + " IS in list at position " + found);
      else
         System.out.println(last + " is NOT in list.");

      last = "Smart";
      found = StudentArrayUtilities.binarySearch(myClass, last, 0,
            myClass.length - 1);
     if ( found >= 0 )
         System.out.println(last + " IS in list at position " + found);
      else
         System.out.println(last + " is NOT in list.");
   }
}
```

```
class Student
{
   private String lastName;
   private int totalPoints;

   public static final String DEFAULT_NAME = "zz-error";
   public static final int DEFAULT_POINTS = 0;
   public static final int MAX_POINTS = 1000;

   // constructor requires parameters - no default supplied
   public Student( String last, int points)
   {
      if ( !setLastName(last) )
         lastName = DEFAULT_NAME;
      if ( !setPoints(points) )
         totalPoints = DEFAULT_POINTS;
   }

   public String getLastName() { return lastName; }
   public int getTotalPoints() { return totalPoints; }

   public boolean setLastName(String last)
   {
      if ( !validString(last) )
         return false;
      lastName = last;
      return true;
   }


   public boolean setPoints(int pts)
   {
      if ( !validPoints(pts) )
         return false;
      totalPoints = pts;
      return true;
   }
```

```
   // could be an instance method and, if so, would take one parameter
   public static int compareTwoStudents( Student firstStud, Student secondStud )
   {
      int result;

      // this particular version based on last name only (case insensitive)
      result = firstStud.lastName.compareToIgnoreCase(secondStud.lastName);

      return result;
   }

   public String toString()
   {
      String resultString;

      resultString = " "+ lastName
         + " points: " + totalPoints
         + "\n";
      return resultString;
   }

   private static boolean validString( String testStr )
   {
      if (testStr != null && Character.isLetter(testStr.charAt(0)))
         return true;
      return false;
   }

   private static boolean validPoints( int testPoints )
   {
      if (testPoints >= 0 && testPoints <= MAX_POINTS)
         return true;
      return false;
   }
}

class StudentArrayUtilities
{
   // print the array with string as a title for the message box
   // this is somewhat controversial - we may or may not want an I/O
   // methods in this class.  we'll accept it today
   public static void printArray(String title, Student[] data)
   {
      String output = "";

      // build the output string from the individual Students:
      for (int k = 0; k < data.length; k++)
         output += " "+ data[k].toString();

      // now put it in a JOptionPane
      JOptionPane.showMessageDialog( null, output, title,
         JOptionPane.OK_OPTION);
   }
```

```
// returns true if a modification was made to the array
private static boolean floatLargestToTop(Student[] data, int top)
{
   boolean changed = false;
   Student temp;

   // compare with client call to see where the loop stops
   for (int k = 0; k < top; k++)
      if ( Student.compareTwoStudents(data[k], data[k+1]) > 0 )
      {
         temp = data[k];
         data[k] = data[k+1];
         data[k+1] = temp;
         changed = true;
      }
   return changed;
}

// public callable arraySort() - assumes Student class has a compareTo()
public static void arraySort(Student[] array)
{
   for (int k = 0; k < array.length; k++)
      // compare with method def to see where inner loop stops
      if ( !floatLargestToTop(array, array.length-1-k) )
         return;
}

public static int arraySearch(Student[] array,
   String keyFirst, String keyLast)
{
   for (int k = 0; k < array.length; k++)
      if ( array[k].getLastName().equals(keyLast) )
         return k;  // found match, return index

   return -1;   // fell through - no match
}
```

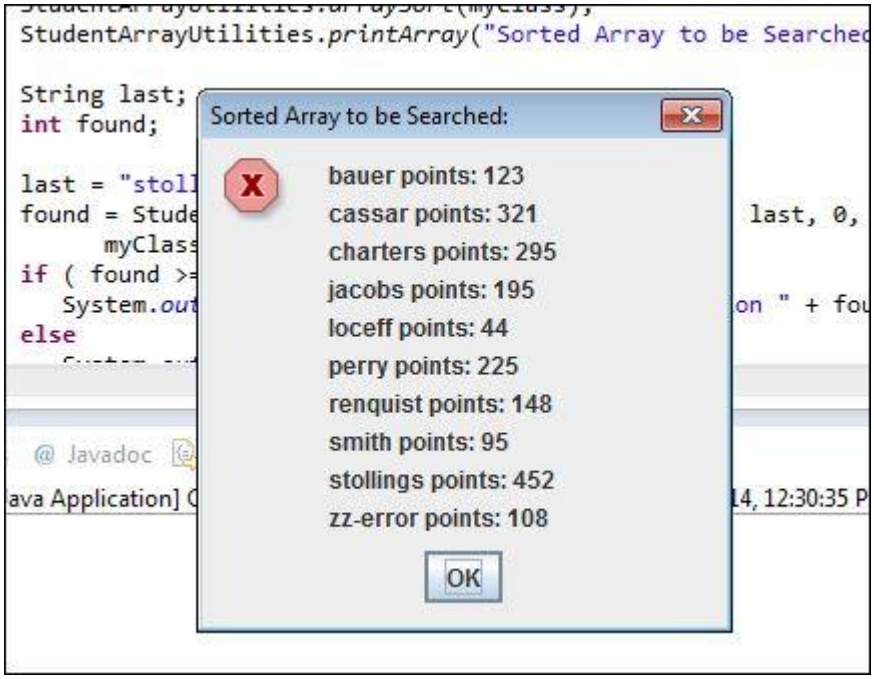
```
   public static int binarySearch(Student[] array, String keyLast,
       int firstIndex, int lastIndex)
    {
       int middleIndex, result;

       if (firstIndex > lastIndex)
          return -1;

       middleIndex = (firstIndex + lastIndex) / 2;
       result = keyLast.compareToIgnoreCase(
          array[middleIndex].getLastName());

       if (result==0)
          return middleIndex;   //found him!
       else if (result < 0)
          return binarySearch( array, keyLast, firstIndex,
             middleIndex-1);
       else
          return binarySearch( array, keyLast,
             middleIndex+1, lastIndex);
    }
}
```

The output:



... then ...