|  |
| --- |
| UMUC CMSC 350 Data Structures and Analysis |
| Implementing Sorting Algorithms |
| CMSC 350 Project #3 |
|  |
| **Megan Tran** |
| **2/19/2016** |

|  |
| --- |
| A Java programming project by Megan Tran. I implement one O(n\*n) sorting algorithm called insertion sort, and one O(nlogn) algorithm called quick sort. I use my test driver class RelayPlayer to test the new sorting methods. |

Megan Tran

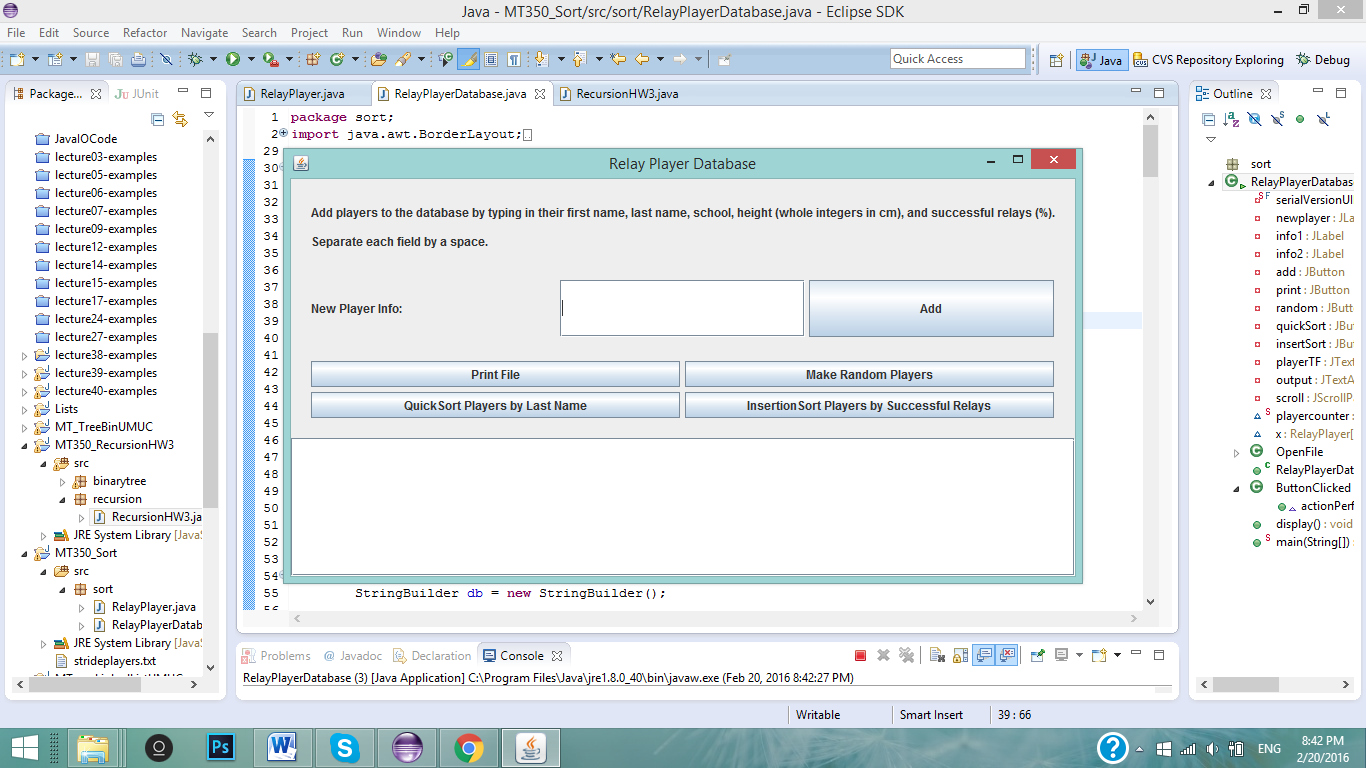
CMSC 350

Project #3 Documentation

Project #3: Sorting Algorithms

1. **Program Description and Guide**

\*Used Java as program code\* All of Project #3 source code and files are filed inside the compressed zip file MT350\_Sort. Project classes are filed under the sort package, and include **RelayPlayer.java and RelayPlayerDatabase.java.** The two Java files are taken from week 1 of class and modifiedfor this project. I also include a textfile of RelayPlayer information called strideplayers.txt. I am implementing one of the O(n\*n) sorting algorithms, and one of the O(nlogn) sorting algorithms. Of the O(n\*n) algorithms, I chose to implement the **Insertion sort**, and for O(nlogn) I chose to implement a **Quick sort**. All new methods can be found in the RelayPlayer class. For further testing and display, I have added two new buttons to the RelayPlayerDatabase GUI for the sorting methods. The new GUI looks like this:



The newest buttons, “QuickSortPlayers by Last Name”, and “InsertionSort Players by Successful Relays” will display completely sorted RelayPlayer arrays inside the text area. QuickSort button displays the RelayPlayers by last name in ascending order, and InsertionSort button displays the RelayPlayers by successful relays in ascending order.

For further testing in the correctness of the sorting algorithm, the main test driver in the RelayPlayer class displays the sorting of an array of RelayPlayer objects after each step of the sort.

For **insertion sort**, I created a new method called **public** **static** RelayPlayer[] **insertionSort**, which takes in a parameter of an array of RelayPlayers, and returns the array, sorted. It uses the insertion sort algorithm to successfully sort the array. For ever iteration, the method will take one element of the unsorted array at one time, and place it at an index where there are no lesser values. It returns a fully sorted RelayPlayer array in the end.

For quick sort, I created two methods – one helper method named **private static** void **swap**, which takes three parameters – a RelayPlayer array, a first index, and a second index. The method swaps the elements of the array in the first index with the element in the second index. This method is used in the **public static** RelayPlayer[] **quickSort** method. This method holds the logic behind the quick sorting algorithm. In the method, a “pivot” element is selected (an element around the middle of the array), in which the method will sort all the elements that are “smaller” than the pivot element to the left, and all elements “larger” than the pivot to the right. Afterwards, **the method recursively calls itself** on the arrays to the left and right side of the pivot element, until the array is fully sorted.

I also created a method called printArray, which takes a parameter of a RelayPlayer array, and prints out the array contents to the console. It is used for testing.

**How to Use**: After unzipping the project folder, all java files can be found under the **sort** package inside MT350\_Sort. Run the files in a Java IDE (Eclipse, Netbeans, etc.). You can run the main method in RelayPlayer for testing the correctness of the two sorting algorithms (Insertion Sort, QuickSort),

Running ReplayPlayerDatabase.java, you can add to and view the list of RelayPlayer objects, and make random players. After making a set of random RelayPlayer objects, you can choose to sort them using QuickSort or InsertionSort by pressing on their buttons. If either sorting buttons are pushed BEFORE creating random players, an error message will pop up.

1. **Program Design**

The program design remains similar to the first homework created for CMSC350, however I modified RelayPlayer.java with three new methods for the sorting algorithms. For the RelayPlayerDatabase GUI, I have added two new buttons, QuickSort and InsertionSort, for their corresponding sorts.

Summary of RelayPlayer methods: A **constructor** that takes in a String to be split into RelayPlayer’s various fields, a **compareTo** method between RelayPlayers, and a **makeRandom** method that makes a random amount of RelayPlayer objects depending on the integer (int N) in the parameter, and **toString** method that nicely formats the different fields of the RelayPlayer into a String. The main method of the class tests several of the methods.

**RelayPlayer.java New Methods**

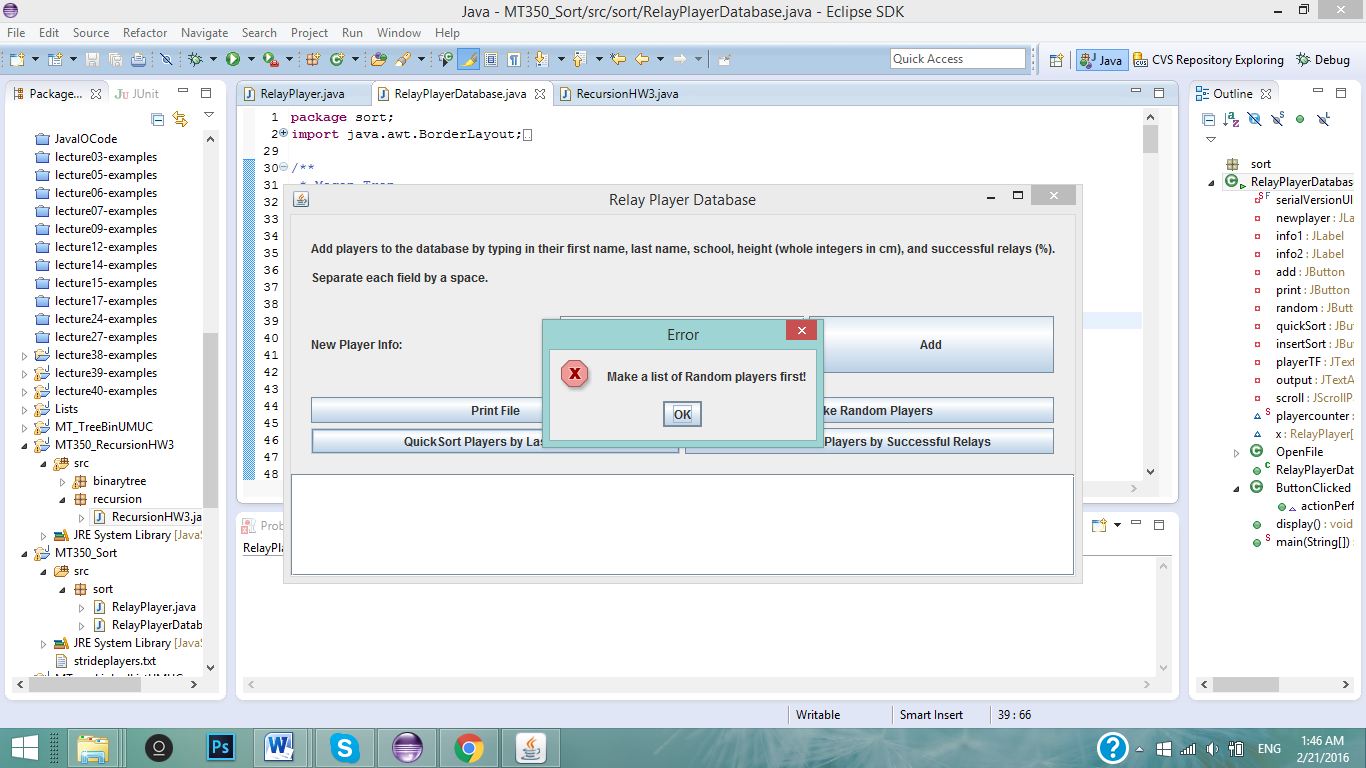
* **public** **static** RelayPlayer[] **insertionSort(RelayPlayer player[])** – performs a insertion sort logic on the parameter array. Returns a fully sorted array.
* **Private static void swap(RelayPlayer[] player, int firstIndex, int secondIndex) –** swaps the element of the player array at the firstIndex with the element at the secondIndex.
* **Public static RelayPlayer[] quicksort(RelayPlayer player[], int front, int end) –** performs quick sort logic on the parameter array. The parameter front represents the beginning index of the array, and the parameter int end represents the ending index of the array. Returns a fully sorted array.
* **Public static String printArray(RelayPlayer player []) –** returns a String of the player array contents. Used for testing purposes.

1. **Testing Plan**

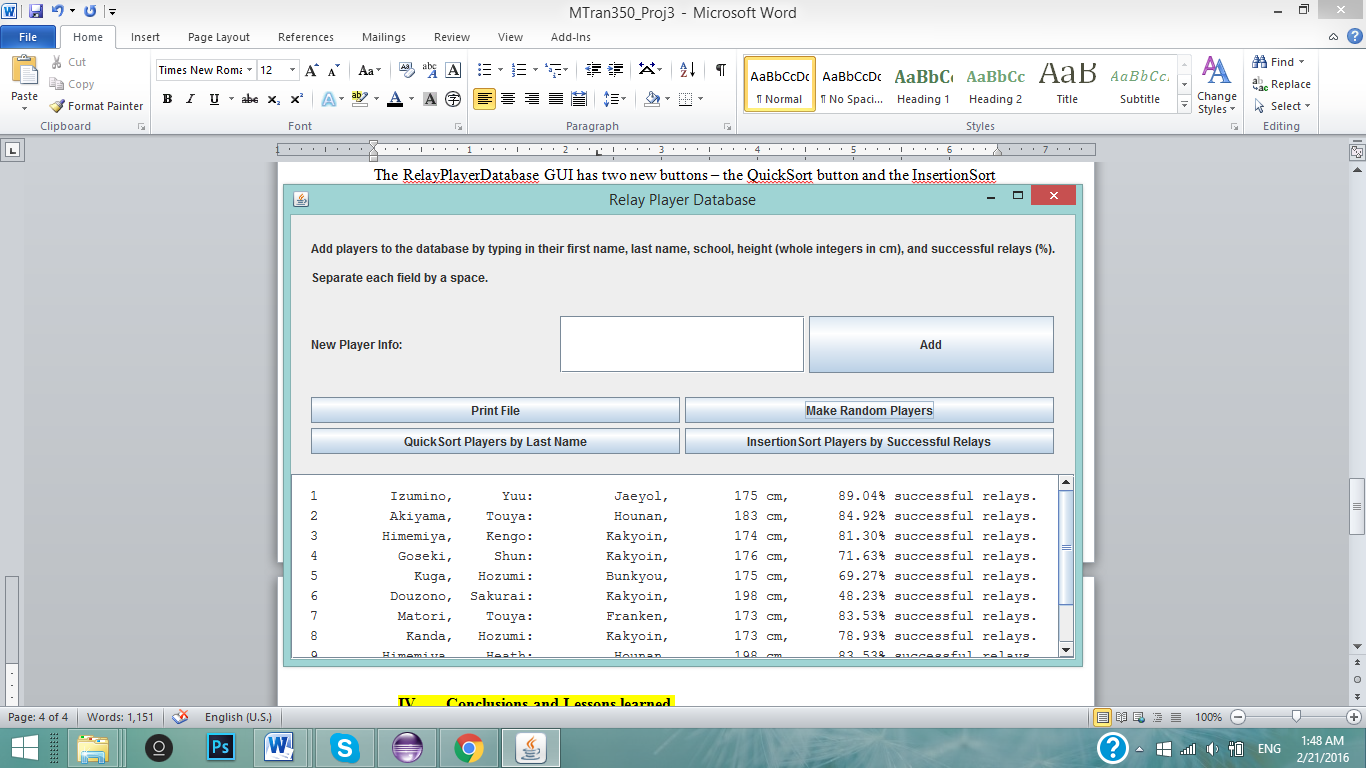
There are two parts to the testing plan. First, I tested the sort methods to make sure they return correctly sorted arrays with the RelayPlayerDatabase GUI. And second, I used the test driver class (RelayPlayer) to take a closer look at how each sort algorithm (quicksort and insertionSort) sorts an array.

**Testing PART 1** – Testing with the GUI.

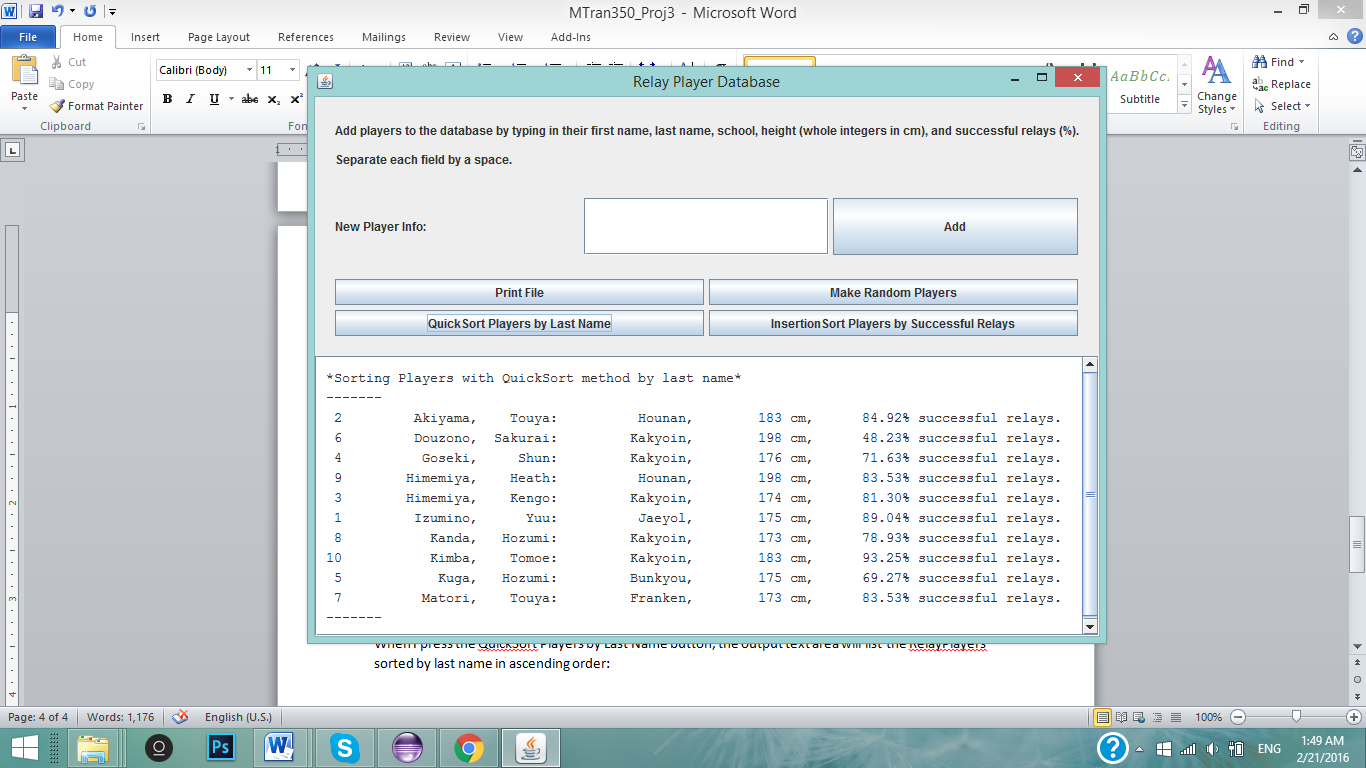
The RelayPlayerDatabase GUI has two new buttons – the QuickSort button and the InsertionSort Button. Each button will sort an array of randomly created players. However, if either button is pushed without the user creating a number of random players, an error message will appear.



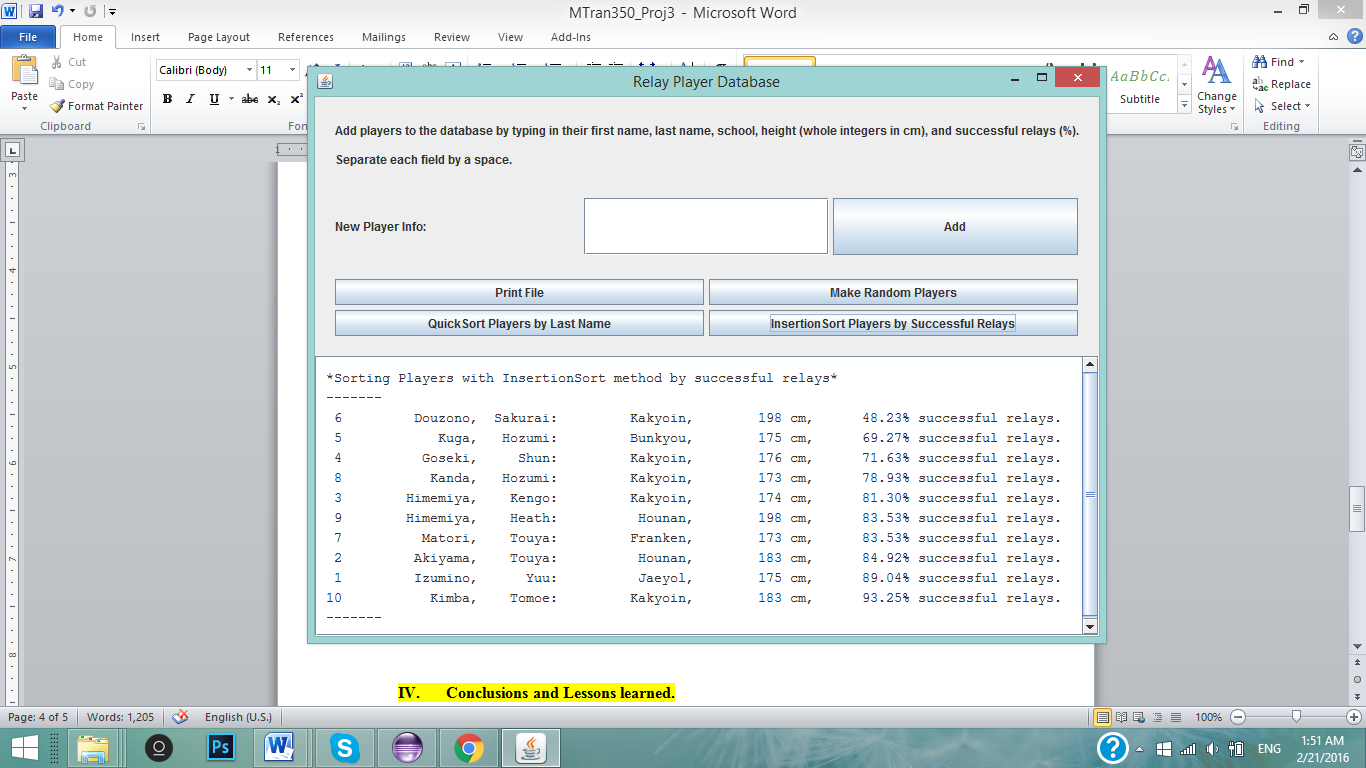
First, I will create a list of 10 random players:



When I press the QuickSort Players by Last Name button, the output text area will list the RelayPlayers sorted (using quicksort) by last name in ascending order:

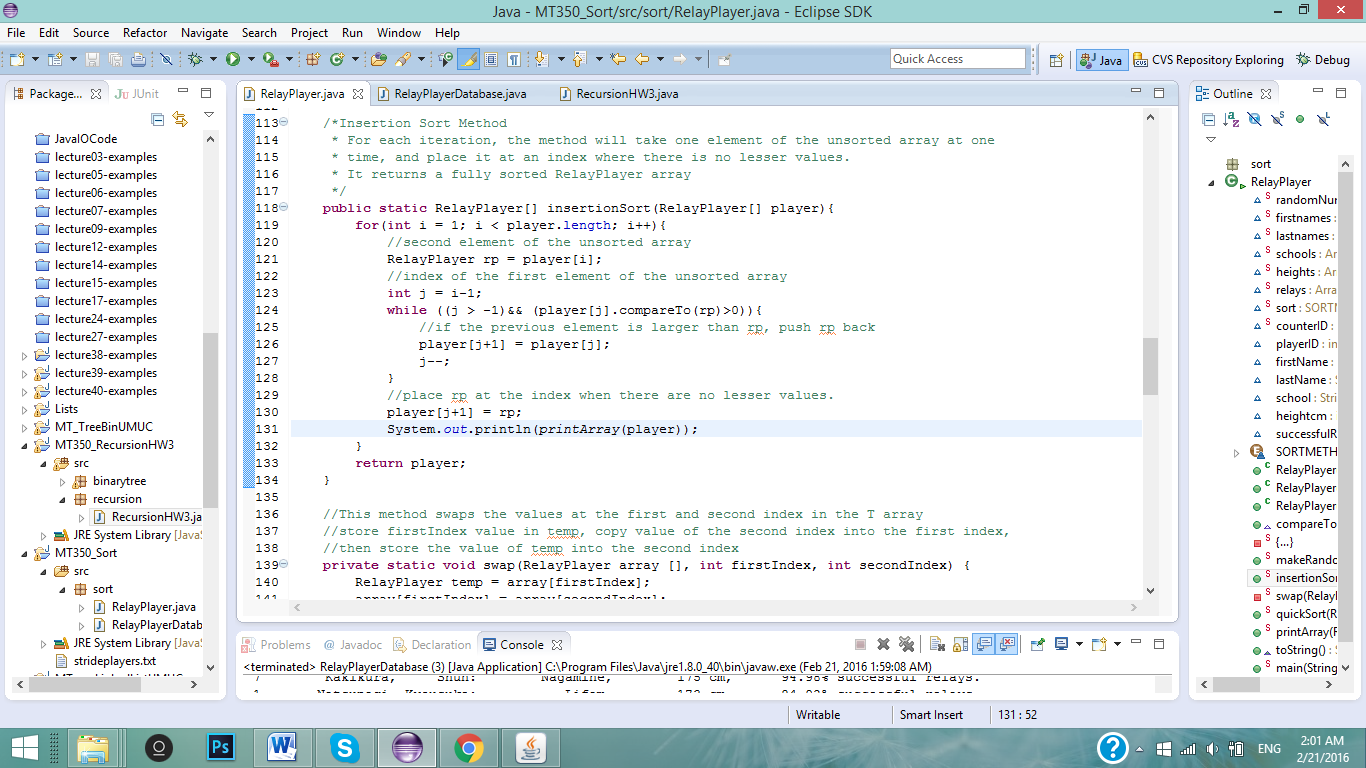


When I press the InsertionSort Players by Successful Relays, the output text area will list the RelayPlayers sorted (using insertionSort) by their successful relays in ascending order:

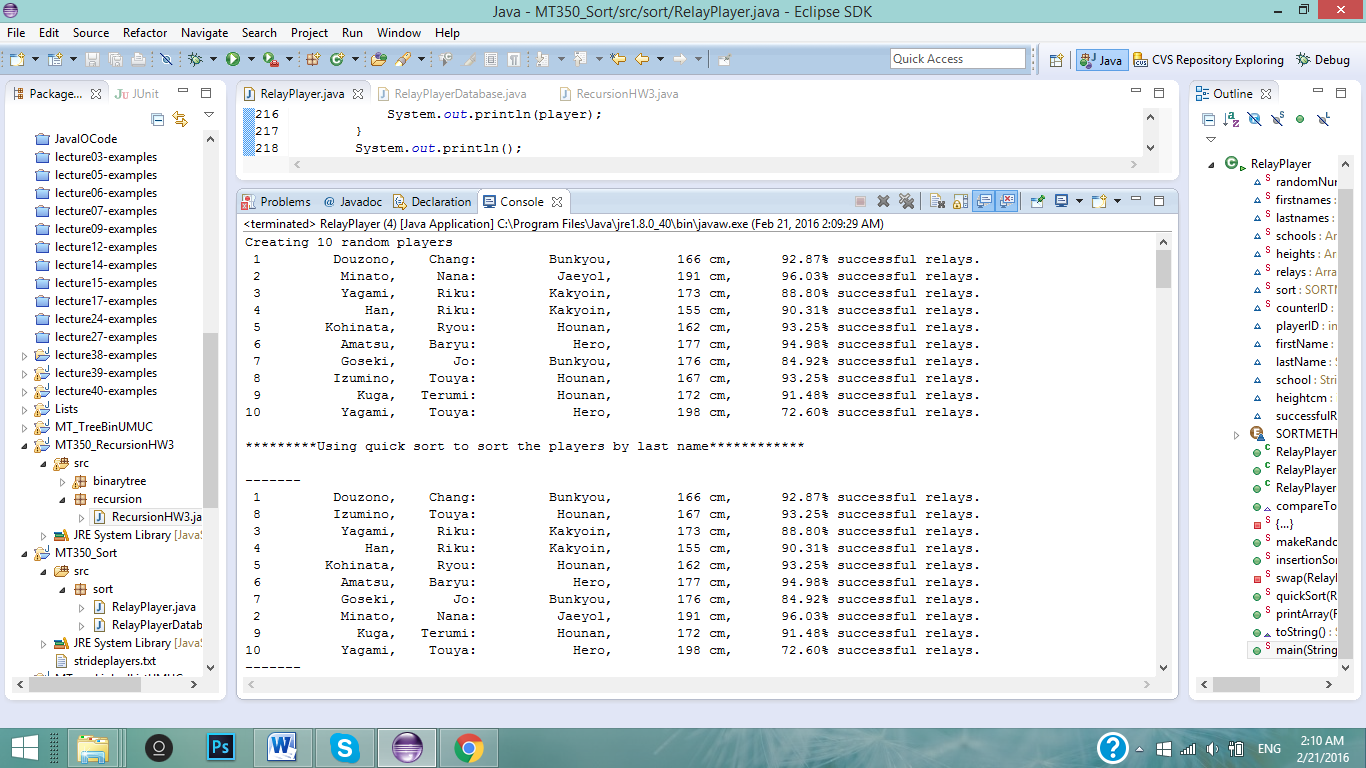


**Testing PART 1** – Testing with the Driver class.

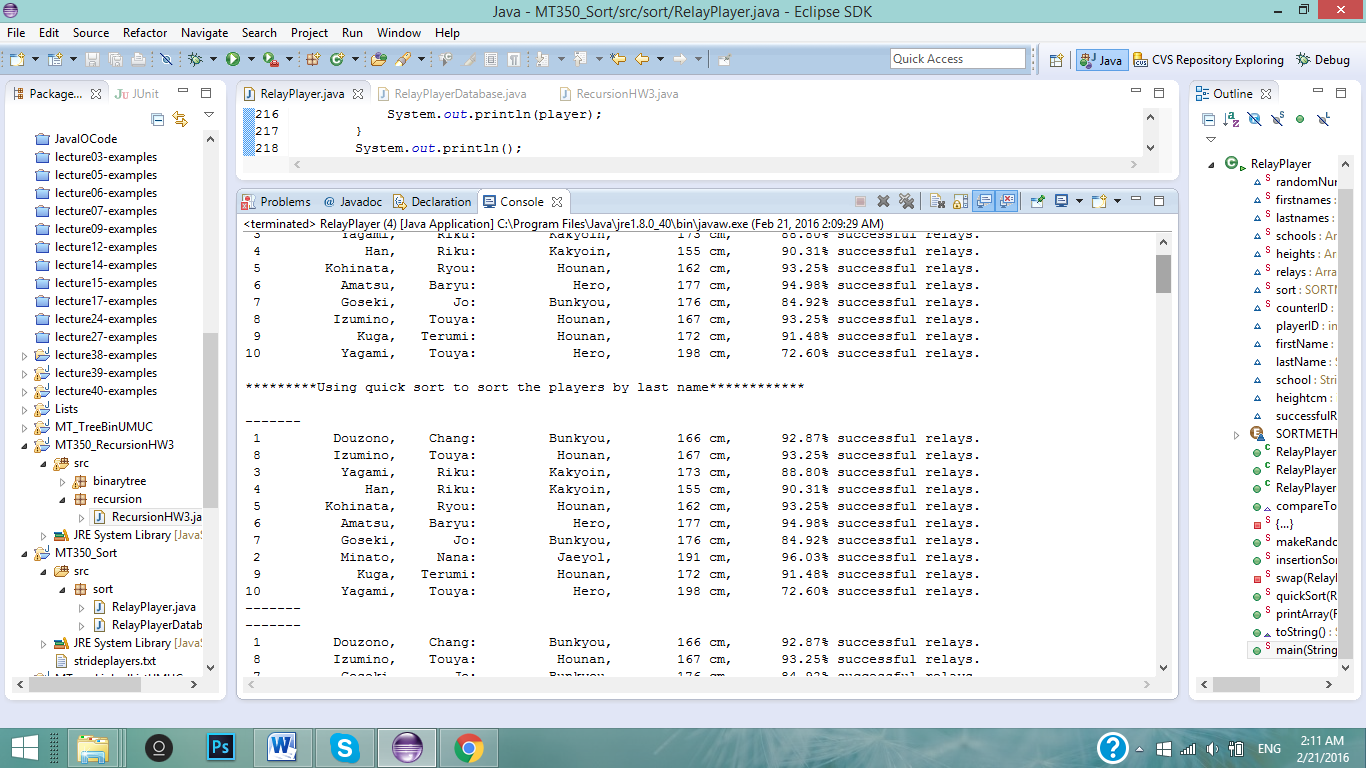
For further testing of the sorting algorithms, I added an extra line of code into each of the sorting methods in order to print out each step (either iteration or call) in the sorting method. I use the **printArray** method to print the array after every step of the sort. For example, in the insertionSort method (highlighted line 131):



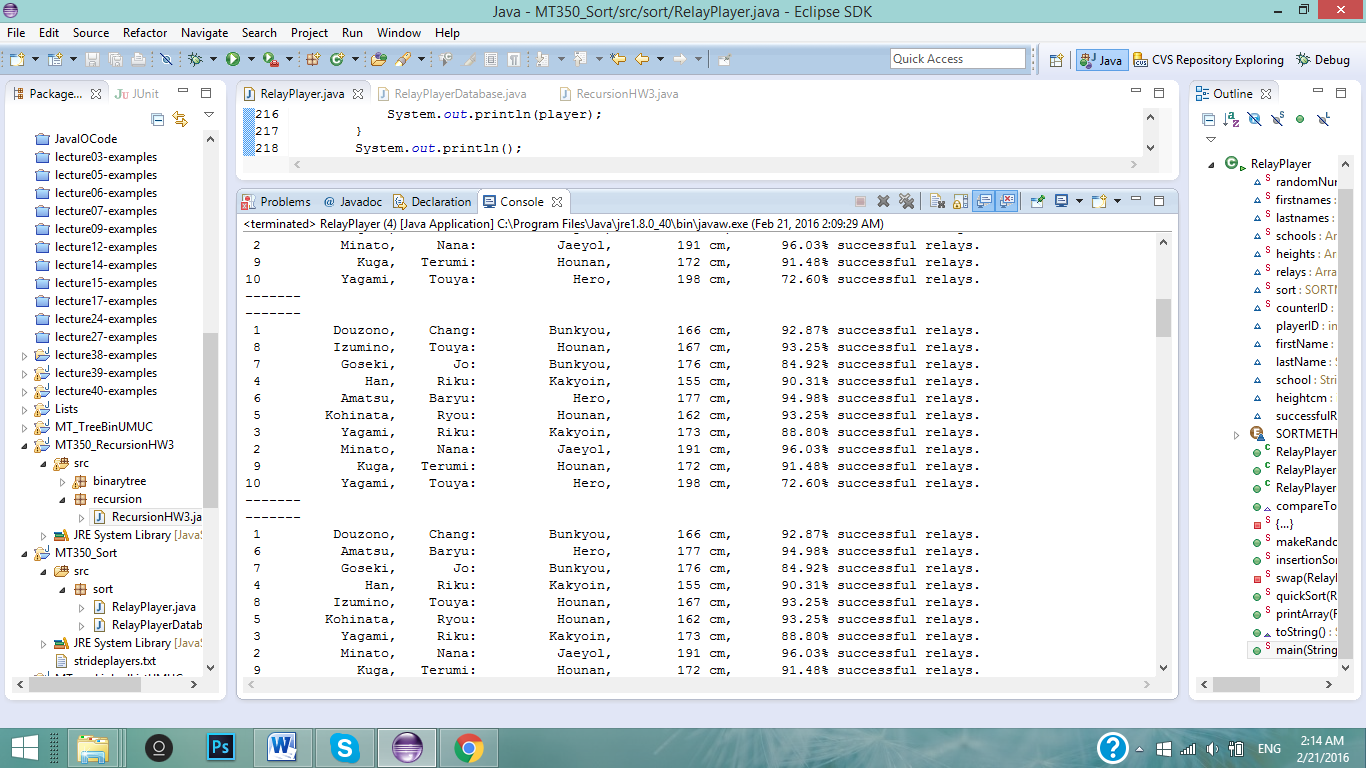
First, I create a set of 10 random RelayPlayer objects:



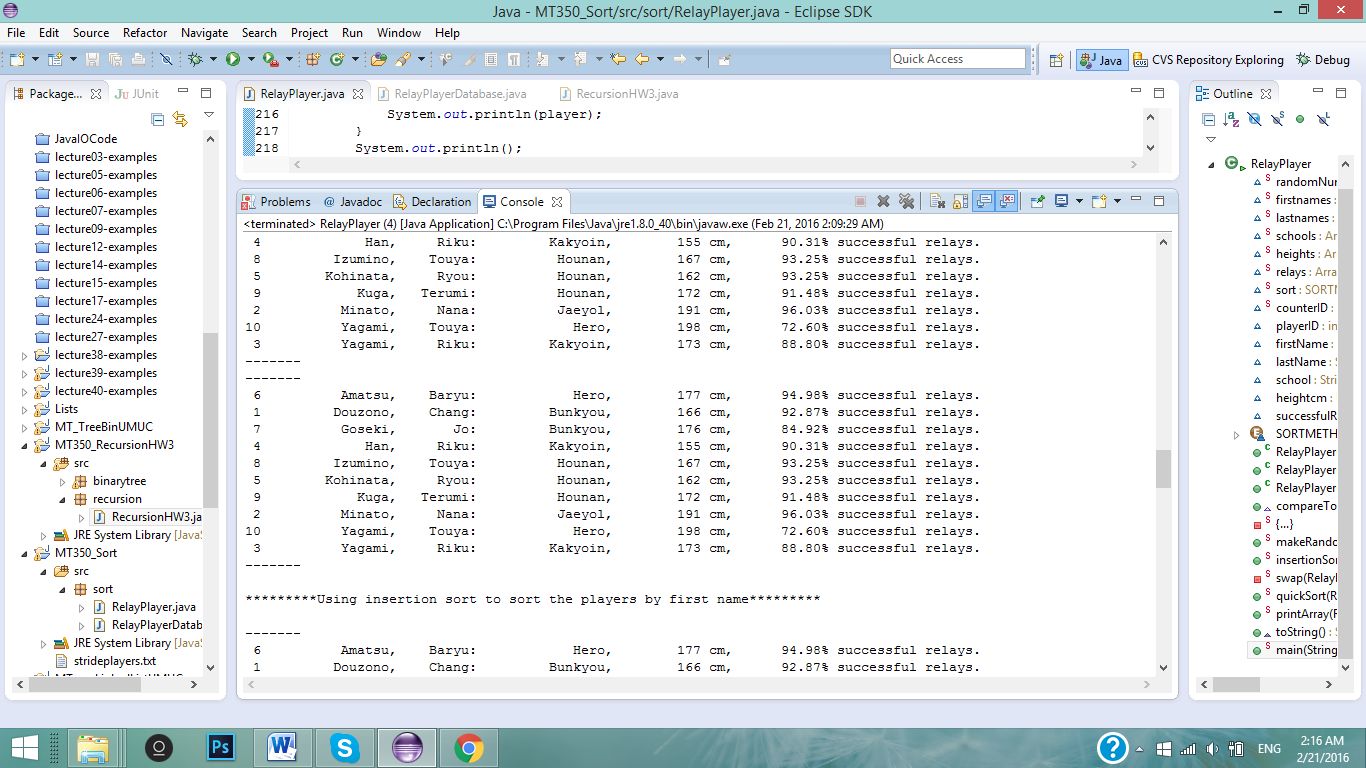
Next, I am going to use the Quick Sort algorithm to sort the list of random players by last name. The pivot element of this test run is RelayPlayer #5 with the last name of **Kohinata**. This means, any last name “smaller” than Kohinata will be sorted to the left, and any last name “larger” will be sorted to the right. After the first iteration, you can see that players #8 (Izumino) and #2 (Minato) have switched places (Minato is larger than Kohinata, while Izumino is smaller than Kohinata.



After a few iterations, the array is sorted so that all last names smaller than Kohinata are on the left, and all last names larger are on the right:



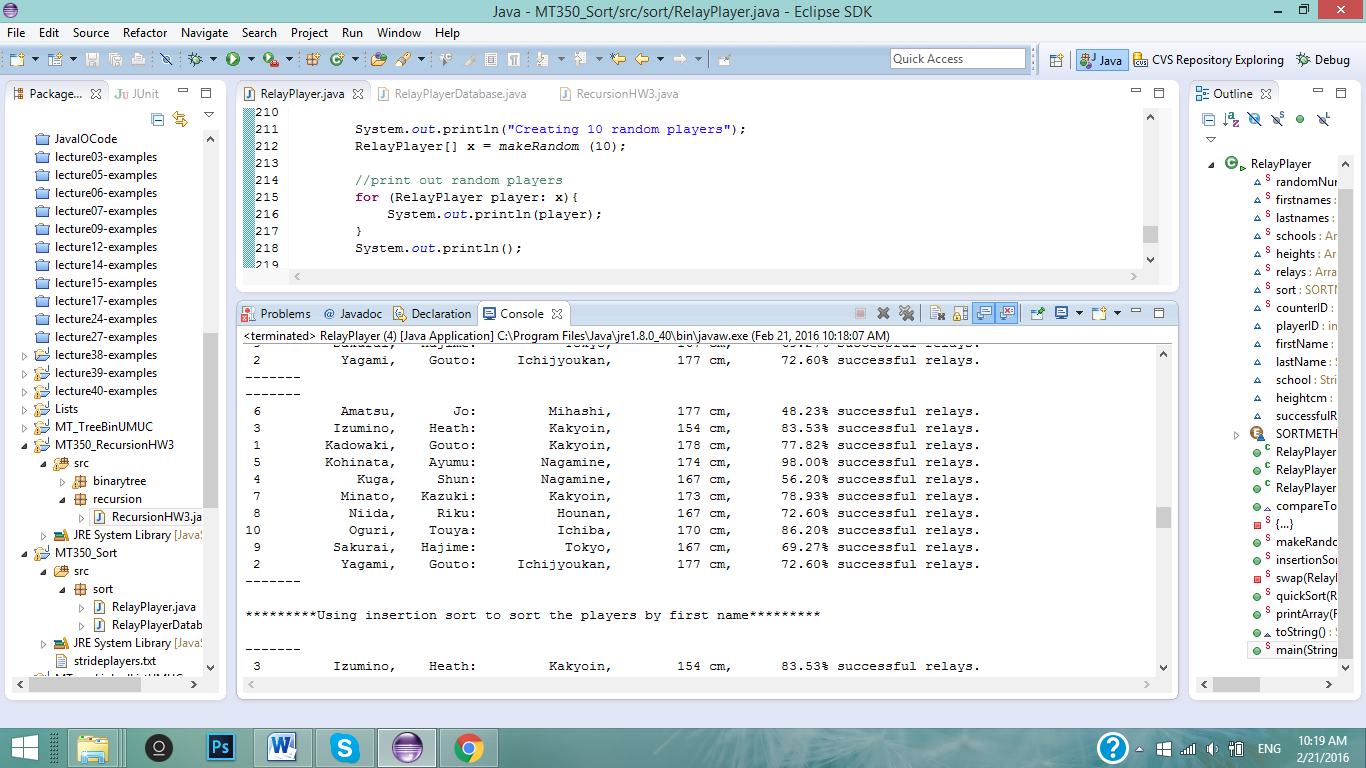
QuickSort method is then called recursively until each side of the array is sorted and in order:

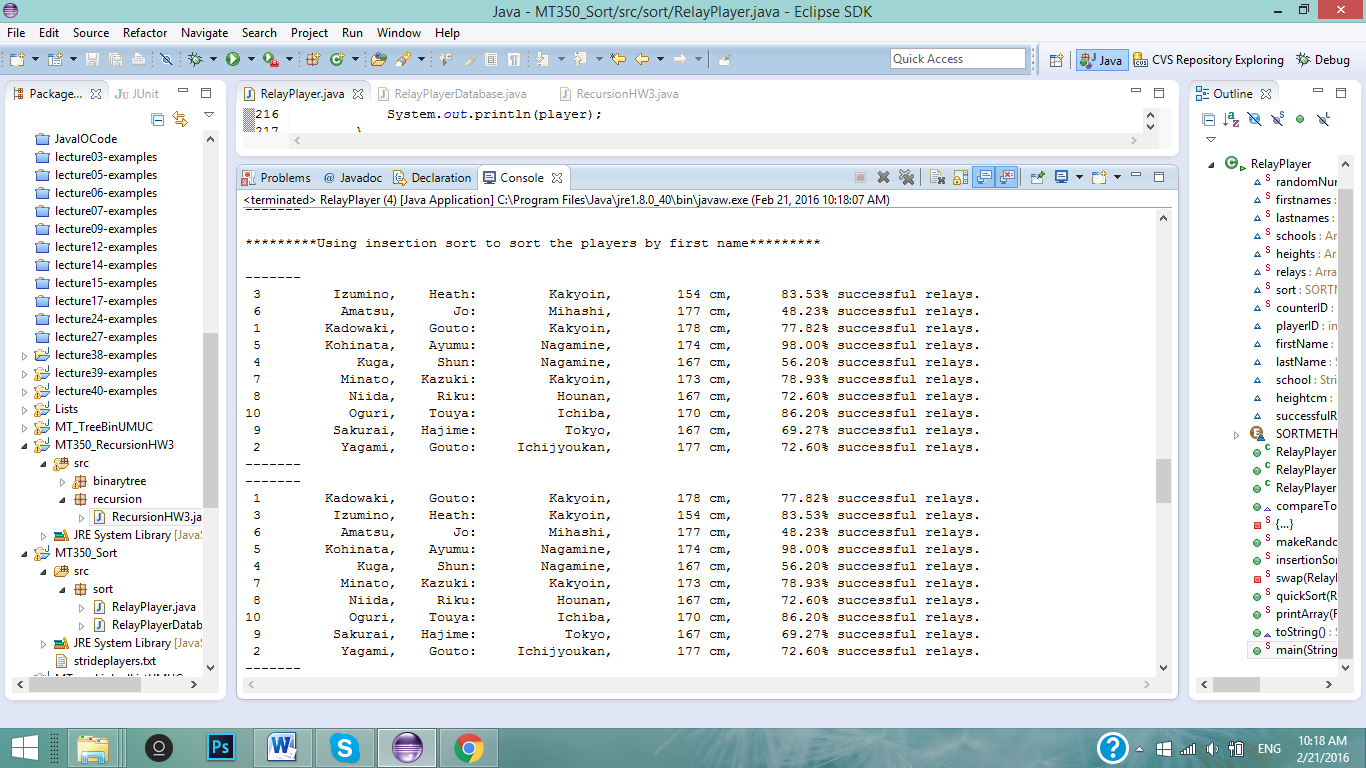


For this test run, the QuickSort method made **11 swaps** until the array was fully sorted at the print out.

For the next test, I sort the same RelayPlayer array using the InsertionSort algorithm, sorting the players in order by their first names. I reran the driver class for a set of players that will help me illustrate the insertion algorithm more easily:

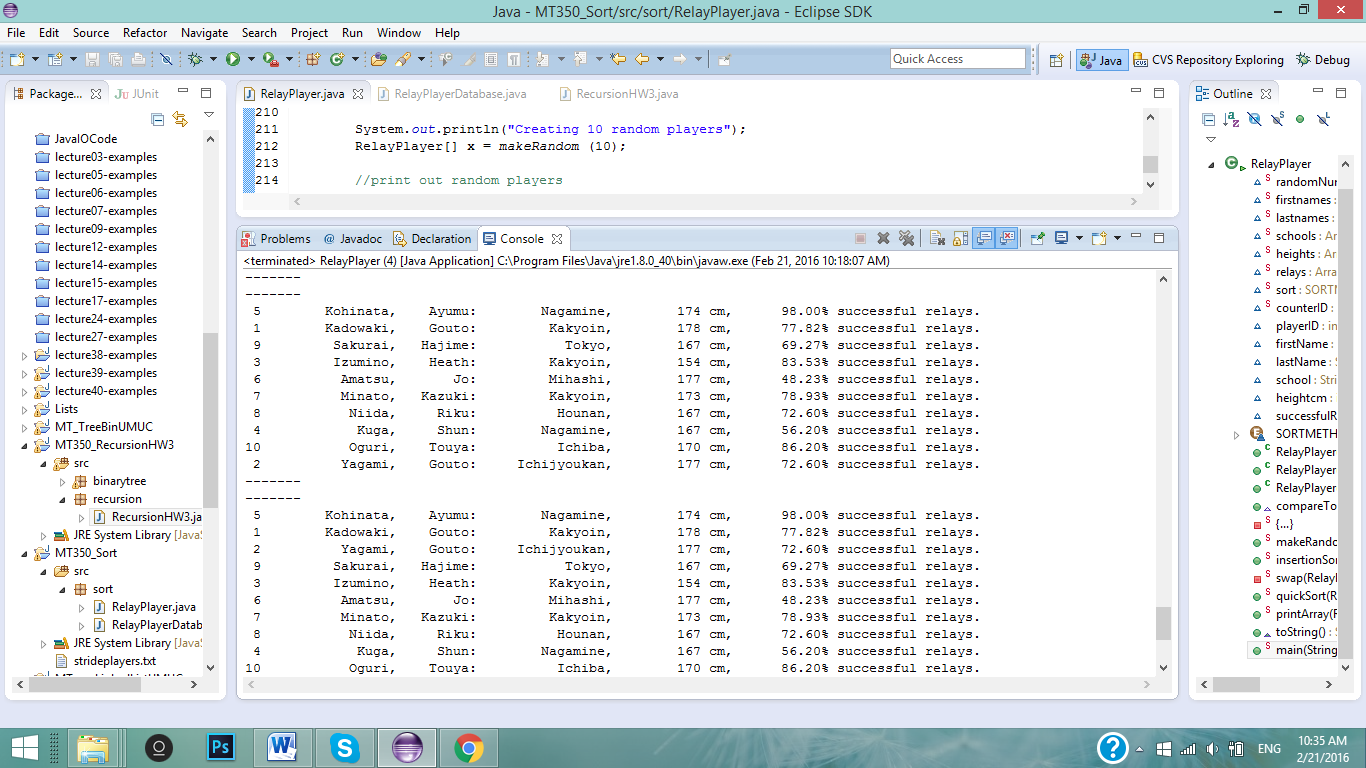
This is the list of players before starting the InsertionSort algorithm.





For the insertion sort algorithm, the method searches for the second element of the array and compares it to the first element. Since the name first element “**Jo**” is larger than the name “**Heath**”, they are switched in the first iteration. The next iteration, the method looks at the next element in the unsorted part of the array – the element with the name “Gouto”. Since Gouto is smaller than both Heath and Jo in the sorted part of the array, Gouto is placed at the front of the array.

The method goes through 9 iterations (the whole array except for the first element), and at the end, the array is fully sorted. This is because the method looks at one element at a time, and sorts it to the index of the array where there are no values less than the element. The below screen shot shows the final iterations and sorted array printed:



1. **Conclusions and Lessons learned.**

When reading about the algorithms in the readings, I thought the logic behind the sorts were quite simple and beautiful. On the other hand, coding them was not so easy. A lesson I learned when writing these sorting methods was to keep track of pointers. With the QuickSort method, I often found myself in the middle of an infinite loop, causing the program to never finish, or to have a stack over flow. I also got caught with the ArrayIndexOutOfBounds exception because I had forgotten to keep track of the end of an array.

Using the printArray method really helped me visualize each step of the sort. I would also be able to see my mistakes for each step of the sort. While I wasfixing all the bugs in my quicksort, I did not understand why my final array was not fully sorted. Using the printArray method, I realized that only half of my array was sorted. I did not sort the array to the right side of the pivot element in my logic. Another great tool I used was the debugger. With the debugger, I was able to find out that I was stuck in an infinite loop, and fix my logic.

In my tests, it seems that the insertion sort algorithm is a bit faster than quick sort. For small data sets like the one I used for tests, for best case, insertion algorithm has a time complexity of O(n), which is quite good for sorting. The quick sort algorithm best case is O(nlogn), but it performs better with larger datasets, while the insertion sort has a complexity of O(n\*n) for average and worst cases.