Criterion C

I chose to build this program in Java Swing. Java Swing is ideal for this project because NetBeans has a built in visual Swing editor ("Designing a Swing GUI in NetBeans IDE"), so learning it is fast and intuitive. Java can also be run on any device with a Java Virtual Machine, so my program would also be portable.

DataManager

I chose to store all user data in .csv files because they are easy to be read by a human, meaning that a human can easily edit the contents of a .csv file. The data is read using the `Scanner` class. Each line of text in the .csv file is read using the scanner, then, depending on which object is required, a new object is created based on the string. I chose to use a `Scanner` object to process the files as `Scanner` can read and process tokens as it iterates through the string. It is simpler than placing each individual token from the string into an array using `.split()` then indexing. A new object is then created based on the tokens read. For example, this line:

Bob	12	MALE	37aa4f50-1584-4a2f-b649-b61f6a291ae3

would be read and passed as:

```
new Profile("Bob", 12, Gender.MALE, "37aa4f50-1584-4a2f-b649-b61f6a291ae3")
```

Storing swim time data is more complicated as the data is three dimensional: the date, the stroke, and the profile. To accomplish this, each swim time is stored in a file that matches its date. For example, the swim times for 14 March, 2015 is stored in "14 3 2015.csv". That way, when when the caller for the `getDateData()` and `storeDateData()` methods specify a date for the swim times, the program knows to look for the files that match the specified date. I find this method more intuitive, simpler, and the data can be stored more easily than by using a three dimensional array.

```
    public static HashMap<Profile, ArrayList<Double>> getDateData(String date)
      ...
                  Scanner scanner = new Scanner(new File("./data/" + date + ".csv"));
```

Statistics

Most of this is pretty simple: most of the statistical formulas are simply expressions that can easily be expressed in Java code. Some of them, however, are more complex. The `sigma` method, for example, evaluates the sum of all real numbers from `start`to `end` over the function `f` via recursion. I chose recursion instead of the more intuitive loop as recursion provided a shorter and more elegant solution without hindering readability.

```
    public static double sigma(Lambda f, double start, double end)
    {
        if (start == end)
        {
            return f.f(start);
        }
        return f.f(end) + sigma(f, start, end - 1);
    }
```

`Lambda` is an interface with a single method `f` that takes a double as input and returns another double as output ("How Do I Define a Method Which Takes a Lambda as a Parameter in Java 8?").

Retrieving data for StatsViewer

This section of the program is made unnecessarily complicated due to the fact that the `getNationalStandards` method used to retrieve the data returns an array list of doubles while the `Stat` methods accept an array of doubles. So I have to convert the array lists to arrays using the `listToArray` method. The `listToArray` method uses a for-loop to transfer each item in the array list to the array.
I used 12 variables. 5 for array lists and 5 for arrays, one per stroke. The other two are an array list and array for all swim times combined. The swim times are loaded, and every value that is not -1 (if it is -1, then `null` is added instead so that every fifth element corresponds to a single stroke. The `null` value is later removed) is added to the array list that stores all swim times. After that, every fifth number is added to an array that holds the data for each stroke:

```
        for (int i = 0; i < allTimes.size(); i += 5) // fill each stroke type with
corresponding element from alltimes
        {
            _freestyle100.add(allTimes.get(i));
            _backstroke50.add(allTimes.get(i + 1));
            _backstroke100.add(allTimes.get(i + 2));
            _breaststroke50.add(allTimes.get(i + 3));
            _butterfly50.add(allTimes.get(i + 4));
        }
```

This works because every 5th item corresponds to a stroke:

1	2	3	4	5	1	2	3	4	5	etc...

I chose to create an array list, fill it up with numbers before converting it into an array instead of directly filling in the arrays because the size that the arrays need to be simply hasn't been calculated yet before the arraylists are filled. I feel that the easiest way to determine the size of the arrays is to simply fill the array lists first then use the array list's size as the size for the corresponding array.

```
        allData = new double[allTimes.size()];
        freestyle100 = new double[_freestyle100.size()];
        backstroke50 = new double[_backstroke50.size()];
        backstroke100 = new double[_backstroke100.size()];
        breaststroke50 = new double[_breaststroke50.size()];
        butterfly50 = new double[_butterfly50.size()];
```

Date

Determing whether a date came after a certain date was surprisingly confusing. Directly comparing the years, then the months, then the days, returning true if the condition was met or continuing to the next test else, did not work, so instead, I converted the years, months, and days into strings, concactenated them, converted them back into integers, then compared those integers to get the result. The main downside to this method is that the years must have the same amount of digits; however, there is no reason to call this method on years less than 1000 or greater than 9999 in the context of this program, not to mention that the program will automatically handle bad dates.

```
    public boolean isInFuture(Calendar c)
    {
        String sday = this.day <= 9 ? "0" + this.day : "" + this.day;
        String smonth = this.month <= 9 ? "0" + this.month : "" + this.month;
        int thisday = Integer.parseInt("" + this.year + smonth + sday);
         sday = c.get(Calendar.DAY_OF_MONTH) <= 9 ? "0" + c.get(Calendar.DAY_OF_MONTH)
: "" + c.get(Calendar.DAY_OF_MONTH);
         smonth = c.get(Calendar.MONTH) + 1 <= 9 ? "0" + (c.get(Calendar.MONTH) + 1) :
"" + (c.get(Calendar.MONTH) + 1);
        int calendarday = Integer.parseInt("" + c.get(Calendar.YEAR) + smonth + sday);
        return calendarday < thisday;
    }
```

Figuring out how to highlight individual cells in the `JCalendar` was also challenging. I eventually found out that using a class that implements the `IDateEvaluator` interface works well ("Java Highlighting Specific Dates in

JCalendar Cell"). I called my class `BlueHighlighter` and for each date that needs to be highlighted in blue, it is added to a list within the class before the `BlueHighlighter` object is added to the calendar. Similarly, `GreenHighlighter` which extends `BlueHighlighter`, is used to highlight dates in green.

Word Count: 842