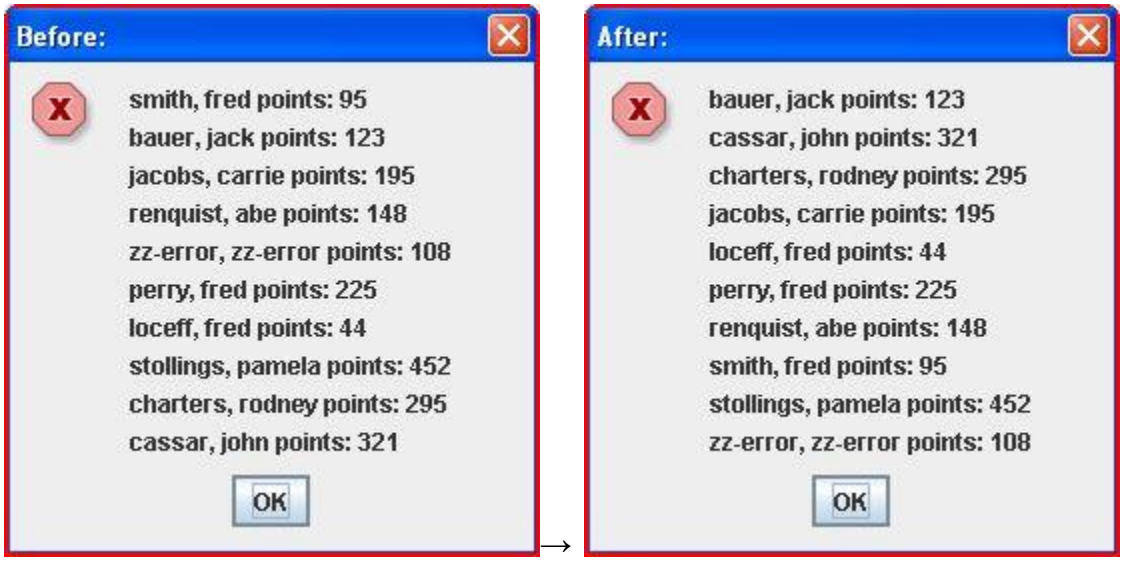
Section 1 - A Student Class

8B.1.1 Adding OOP to Arrays

We've had some interesting examples with arrays and also some fun with classes, but we haven't put them together. This is going to be a very interesting module for you, because we will place these two important ingredients into the same pot and light a match.

Let's start by expanding our student array example so that students have first and last names, and also total scores. Here is an example of a program run. Our job will be to use our OOP wisdom and our knowledge of arrays and sorting to make this happen.



Let's figure out what we can tell about this program and what it must be doing.

- The "After" list is sorted in alphabetical order on last name.
- The point totals for each student have nothing to do with the sort.
- Each student is on his or her own line.
- There is an odd student whose name is "zz-error, zz-error". We can conclude that the client tried to put some illegal name value into a student record and the program replaced the name. Why do you think the string "zz-error" was chosen?
- Whatever this array is, it is not going to be an array of primitives or **Strings** because we see that each element contains both numbers and **Strings**; this is an array of some sort of strange entity. We'll call this bizarre object a "**Student**".

8B.1.2 The Student Class, Introduced

Let's take a stab at the **Student** class. We begin, as always, by defining the private data and supplying a ***constructor***.

Copyright(c) 2006-2014 - Michael Loceff

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

   // constructor requires parameters - no default supplied
   public Student( String lst, String fst, long pts)
   {
      lastName = firstName = "zz-error";
      totalPoints = 0;
      // at least require that it start with a letter
      if (lst != null && Character.isLetter(lst.charAt(0)))
         lastName = lst;
      if (fst != null && Character.isLetter(fst.charAt(0)))
         firstName = fst;
      if (pts >= 0 && pts <= 1000)
         totalPoints = pts;
   }

  // more methods, but what should they be ... ?
}
```

Warning - This is not how we will end up defining the constructors and data. It is just a rough design phase to help us get a feel for the class. We will be adding lots of symbolic statics for min values, max values and defaults, as well as some static validation methods.

It's pretty easy, right? We have an int, **totalPoints**, two **String**s, **lastName** and **firstName**, and some method prototypes with the constructor definition shown. Now we can see where the **zz-error** comes from.

So the client can create **Student *objects*** by supplying the initial data right in the instantiation, as in:

```
Student starPupil = new Student("smith", "frederick", 250);
```

Because we are going to be declaring an ***array of students***, this is not, in fact, how we will ***instantiate*** the students, but we could have done this if we wanted an individual student identified by the reference **starPupil**.

Also, we are not going to supply a **default constructor**. This is very important. It means we cannot instantiate an object like this:

```
Student lateAdd = new Student();
```

Whenever we create a **Student** we must supply the data. The omission of the default constructor is intentional, and one of its implications is that the client must instantiate a **Student** by supplying arguments such as *name* and *score*. Not providing a default (i.e., no parameter) constructor is often done if, for instance, we do not want to supply any other mutator method. It guarantees that all objects of the **Student** class get data at the start through the constructor.

That's a good introduction to this class. Let's peek further at our idea of a main program.

Copyright(c) 2006-2014 - Michael Loceff

Section 2 - Anonymous Objects and a Support Class

8B.2.1 Objects Without Names

We still want an easy way to create an array of **Student** objects. We haven't finished defining the **Student** class yet, but already from what we know about the constructor, we need to supply three pieces of data for each object. And we're trying to do something analogous to what we did with **doubles** and **Strings** earlier:

```
double[] myArray = {10.2, 56.9, -33, 12, 0, 2, 4.8, 199.9, 73, -91.2};

String[] myArray =
{
   "martin", "claudia", "sandra", "samuels",
   "terry", "jack", "clark", "palmer", "abraham", "Mike"
};
```

But how can we do this when the base data type of the class we are using in the array is a class? We can create ***anonymous objects*** in the initialization. An anonymous object is one that has no reference or identifier associated with it. How could such an object even exist? Easy. Look:

```
Student[] myClass =
{
   new Student("smith","fred", 95),
   new Student("bauer","jack",123),
   new Student("jacobs","carrie", 195),
   new Student("renquist","abe",148),
   new Student("3ackson","trevor", 108),
   new Student("perry","fred",225),
   new Student("loceff","fred", 44),
   new Student("stollings","pamela",452),
   new Student("charters","rodney", 295),
   new Student("cassar","john",321),
};
```

Of course. We don't really need any reference identifiers since the ***array name*** and the ***index***, as in **myClass[3]**, are going to be used if we have to identify any of our array elements. So this use of the constructor, in which there is no reference on the LHS of some assignment operator, is perfect.

Copyright(c) 2006-2014 - Michael Loceff

Incidentally, we can also create ***anonymous objects*** if we want to make one just to send off to a method in an argument list, but don't need the object after the call. For example:

```
DrawDot( 100, 40, new Color(255, 255, 128) );
```

This fictitious method call invokes a hypothetical ***anonymous* Color** object which we create just to send to the **DrawDot()** method. Perhaps we are drawing a tan colored dot at location 100, 40 on the screen.

We can also use ***anonymous*** objects to throw together an object to return in a function:

```
public Student makeGuestStudent()
{
   return new Student("Guest", "Student", 0);
}
```

Back to our main topic, though, the initialization of our **Student** array. We can now see why one of the students ended up with the name "zz-error". Look closely at the data and also the constructor.

8B.2.2 A Support Class for Students - StudentArrayUtilities

We add another component to our design: a class designed to process arrays of **Student**s. This will separate the management of *individual* students (a **Student** class responsibility) with that of handling *arrays* of students (our new **StudentArrayUtilities** class). This new class has no data, but does have some static methods. Let's look at the methods:

Copyright(c) 2006-2014 - Michael Loceff

```
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)
   {
      // details omitted here - shown in full listing, shortly
   }

   // 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;
   }
}
```

We see two public methods:

1. **printArray()** - a method that takes an array of **Student** objects and sends it out to the screen, and
2. **arraySort()** - a method that takes an array and rearranges it, leaving it in a sorted order.

For our demonstration, we will only do sorts on last name.

Disclosure. This class has too much user-interface assumption built-in. Specifically, the **printArray()** is going to a **JOptionPane** rather than a ***console*** or something else. It would be better if it were a **toString()** method that the client then sent to its output however it wished. But, we'll leave that for an imagined refinement, since we are short on time and have lots of other topics to cover.

Copyright(c) 2006-2014 - Michael Loceff

8B.2.3 The Client View of the Student and Utility Classes

Now let's look at the entire main class to see how we are going to have to finish defining **Student** and **StudentArrayUtilities**:

```
public class Sample
{
   public static void main (String[] args)
   {
      Student[] myClass =
      {
         new Student("smith","fred", 95),
         new Student("bauer","jack",123),
         new Student("jacobs","carrie", 195),
         new Student("renquist","abe",148),
         new Student("3ackson","trevor", 108),
         new Student("perry","fred",225),
         new Student("loceff","fred", 44),
         new Student("stollings","pamela",452),
         new Student("charters","rodney", 295),
         new Student("cassar","john",321),
      };

      StudentArrayUtilities.printArray("Before: ", myClass);
      StudentArrayUtilities.arraySort(myClass);
      StudentArrayUtilities.printArray("After: ", myClass);
   }
}
```

We're good with the ***array initialization***. Beyond that, there are only three method calls, just like the earlier examples. However, what's interesting about these calls is that they have a class name in front of them. What do you know about method calls ***dereferenced*** by class names? I'm waiting ... .

Correct. ***Static class methods***.

And why do we use ***static class methods***, ever? Two reasons, both applicable here:

1. They involve data or objects of their own class or a related class, and
2. they do not make sense to be called from any individual object.

Great. This tracks. Printing or sorting an ***array*** of **Students** is certainly something that should be done by the **Student**-related class like **StudentArrayUtilities**, since such a class would know all the details and pitfalls of the **Student** data that it supports. At the same time, printing or sorting an array of **Student**s means dealing with a whole group. There is no one single **Student** ***object*** that would naturally be used to ***dereference*** this kind of method. Nor do we need an object of **StudentArrayUtilities**. That's why we are going to declare these as ***static methods***, not as ***instance methods.***

Copyright(c) 2006-2014 - Michael Loceff

This is so important, I can't say it loud enough. You have to understand why we are going to use a ***static method,*** not an ***instance method***, for these two operations. Also, in these cases, we are passing an ***array of*** **Student** objects as an argument to the methods **,** but in other situations we may not pass *any* objects of the class into the ***static methods***. It depends on what the static methods are designed to do. For instance the **Math.pow()** static method does not take **Math** objects - there are no such things. It takes two doubles. And the **Character.isLetter()** method does not take a **Character** object, but rather a **char** value.

Let's take a peek at another static method (which we have not seen yet), this one in the **Student** class itself:

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

Why is this method static? The reason is that it does not act directly on any **Student** member. The **String** it is testing for validity is not associated with a particular member -- it is used by *two* members **firstName** and **lastName** (as we shall see). Its job is to check whether the **String** that is potentially being assigned to one of those two private members is legal for the class.

8B.2.4 The Full Student Class Definition

Here's another static method of the **Student** class. This one is going to be used by the utility class's **arraySort()** method; it establishes the basis for the sort: last name.

```
   // 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;
   }
```

The comment at the top indicates that it didn't *have* to be a static method, and if it were designed to be an instance method, instead, it would take only *one* parameter (*why, and what parameter?*). However, there is something balanced about seeing both objects to be compared as parameters.

Static class methods can take any kind of parameter that makes sense for what they do. The only unusual and probably unlikely parameter they would take would be a single object of the class itself. Does anyone want to tell me why that would probably be a bad or at least rare, situation? Public forum, please.

Copyright(c) 2006-2014 - Michael Loceff

So, we are ready to see the rest of our **Student** class.

```
class Student
{
   private String lastName;
   private String firstName;
   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, String first, int points)
   {
      if ( !setLastName(last) )
         lastName = DEFAULT_NAME;
      if ( !setFirstName(first) )
         firstName = DEFAULT_NAME;
      if ( !setPoints(points) )
         totalPoints = DEFAULT_POINTS;
   }

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

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

   boolean setFirstName(String first)
   {
      if ( !validString(first) )
         return false;
      firstName = first;
      return true;
   }

   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
   static int compareTwoStudents( Student firstStud, Student secondStud )
   {
      int result;
```

Copyright(c) 2006-2014 - Michael Loceff

```
      // 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
         + ", " + firstName
         + " 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;
   }
}
```

Study this class first, before going on.

Copyright(c) 2006-2014 - Michael Loceff

8B.2.5 The Full StudentArrayUtilities Class Definition

```
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;
   }
}
```

Copyright(c) 2006-2014 - Michael Loceff

Section 3 - The Student Class, Completed

8B.3.1 The Full Listing

I've said so much about this, there isn't anything to do now other than show you the source. If you have any questions, you know how to reach me.

```
import javax.swing.*;

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

      StudentArrayUtilities.printArray("Before: ", myClass);
      StudentArrayUtilities.arraySort(myClass);
      StudentArrayUtilities.printArray("After: ", myClass);
   }
}

class Student
{
   private String lastName;
   private String firstName;
   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;
```

Copyright(c) 2006-2014 - Michael Loceff

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

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

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

public boolean setFirstName(String first)
{
   if ( !validString(first) )
      return false;
   firstName = first;
   return true;
}

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

Copyright(c) 2006-2014 - Michael Loceff

```
   // 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
         + ", " + firstName
         + " 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);
   }
```

Copyright(c) 2006-2014 - Michael Loceff

```
   // 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;
   }
}
```

8B.3.2 P.S.

Aha. I thought of something I could say.

Notice how we have modularized the preparation of an individual **Student String** output in a **toString()** method of the **Student** class and used that in the **printArray()** method of the utility class.

Another afterthought/question: Why did I (could I) declare **floatLargestToTop()** private?

I am very excited for you at this point, because if you understand this example, you are really getting a serious dose of object-oriented programming. This nicely demonstrates the infinity of possibilities that arise when we start using our noodle to solve problems using classes, arrays and other computer programming language concepts. Everything you learn here is applicable to C++, C# or any other object-oriented language.

Let's look at a completely different possibility, next.

Copyright(c) 2006-2014 - Michael Loceff

Section 4 - Arrays Inside Classes

8B.4.1 ArrayIndexOutOfBoundsException

We want to count the number of occurrences of the letters **'A'** through **'Z'** in a **String** that the user types. We might use an array of longs that has 26 elements. **elem[0]** would hold the count of **'A's**, **elem[1]** the count of **'B's,** ... and **elem[25]** the count of **'Z's.**

However arrays create a big headache for us. Look:

```
long[] badFreq = new long[26];

badFreq[-7] = 10;
badFreq[50] = 1;
badFreq[5] = -3;
```

The first two are out-of-bounds of the array. We can only access elements 0 through 25. If we try to run this we get:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
   -7 at Sample.main(Sample.java:12)
```

The program "crashes." One solution is to ***catch*** an **Exception**, as we learned to do with **NumberFormatException**. We use the same form as we did before, but this time catch the **ArrayIndexOutOfBoundsException**.

```
try
{
   badFreq[-7] = 10;
   badFreq[50] = 1;
   badFreq[5] = -3;
}
catch (ArrayIndexOutOfBoundsException e)
{
   // deal with the problem of index out of bounds
}
```

But this causes our program flow to become interrupted. Wouldn't it be better if the out-of-bounds problems were not so disruptive? Also, there is no such thing as a negative count when it comes to number of occurrences of a letter; we would like the last assignment, **badFreq[5] = -3**, to be declared illegal, something not currently done.

We can fix all of this by creating a class called **Frequency** which ***wraps*** around the ***array of longs,*** and protects against all of this ugliness.

Copyright(c) 2006-2014 - Michael Loceff

8B.4.2 A Robust Array: Frequency

We ***wrap*** our ***array of longs*** in a new class called **Frequency**. Since we want **Frequency** to be flexible, we won't restrict it to 26 elements, but allow the number of elements to be set during ***instantiation*** by the ***constructor***.

```
class Frequency
{
   private long count[];
   private int size;
   private static int MAX_SIZE = 10000;

   public Frequency(int size)
   {
      if (size < 0 || size > MAX_SIZE)
         size = MAX_SIZE;

      this.size = size;
      count = new long[size];
   }

   public long get(int index)
   {
      if (index >=0 && index < size)
         return count[index];
      else
         return -1;
   }
   public void increment(int index)
   {
      if (index >=0 && index < size)
         count[index]++;
   }

   public void decrement(int index)
   {
      if (index >=0 && index < size)
         if (count[index] > 0)
            count[index]--;
   }
}
```

This class provides methods for incrementing and decrementing the individual elements of the array, and also an accessor to return any value in the array. It provides the following protections:

- No direct public access to the frequency counts.
- No attempts to increment or decrement an index outside the bounds of the array.
- No accidental decrements below 0, since counts below 0 have no meaning.

Copyright(c) 2006-2014 - Michael Loceff

Here is a nice use of this class:

```
public class Sample
{
   public static void main (String[] args)
   {
      int k;
      Frequency letters = new Frequency(26);

      // this block should leave a 27 in letter[2]
      for (k = 0; k < 28; k++)
         letters.increment(2);
      letters.decrement(2);

      // this block should leave a 59 in letter[25]
      for (k = 0; k < 59; k++)
         letters.increment('Z' - 'A'); // this is 25

      // some illegal accesses
      letters.decrement(500);
      letters.increment(-3);

      // display whole table, going "too far"
      for (k = -3; k < 30; k++)
      {
         // every 5 items, generate a newline
         if ( k % 5 == 0)
            System.out.println("");

         System.out.print( k + ": " + letters.get(k) + "      ");
      }
   }
}
```

This gives us the following console output:

```
-3: -1      -2: -1      -1: -1
0: 0      1: 0      2: 27      3: 0      4: 0
5: 0      6: 0      7: 0      8: 0      9: 0
10: 0      11: 0      12: 0      13: 0      14: 0
15: 0      16: 0      17: 0      18: 0      19: 0
20: 0      21: 0      22: 0      23: 0      24: 0
25: 59      26: -1      27: -1      28: -1      29: -1
```

As you can see, the value -1 is sent back to the client indicating a bad index access. No exceptions are thrown when we run this program because our Frequency class is robust and handles the errors with alacrity.

Also, notice that we are using **char** notation as a more readable option to access the elements. Instead of trying to figure out what index the letter **'R'** refers to, we use the numeric expression **'R' - 'A',** which automatically converts the **'R'** ***ASCII code*** to the proper index. Obviously, **'A' - 'A'** becomes **0,** as it should, **'B' - 'A'** becomes **1**, etc. So, instead of passing integer indexes directly, we will take a **char, letter,** and form the index using the expression **letter - 'A'**.

Copyright(c) 2006-2014 - Michael Loceff

We are only going to have to deal with capital letters (upper case) in our program, as you will see.

Section 5 - A Character Counter Class

8B.5.1 Classes Using Other Classes

Now that we have our **Frequency** class, we can use it inside another class, **CharacterCounter**, that we are going to build. **CharacterCounter** is going to do almost everything for us, and the **main()** method won't need to even know about the **Frequency** class.

CharacterCounter will have two members:

- A **String** that is being analyzed, and
- A **Frequency** object that holds the count for that **String**.

```
class CharacterCounter
{
   private Frequency letters;
   private String userString;

   // ... methods for class ...

}
```

In our client, we will get a **String** from the user and use it to instantiate a **CharacterCounter** object.

```
   Scanner input = new Scanner(System.in);

   System.out.println("Enter a phrase or sentence: ");
   String userPhrase = input.nextLine();

   // create a CharacterCounter object for this phrase
   CharacterCounter freq = new CharacterCounter(userPhrase);
```

All of the counting is done inside the **CharacterCounter** constructor so by the time we finish instantiating this object, we are done! All we have to do is print out the results. We will give **CharacterCounter** an accessor method, **getCount()** that will return the count of individual letters. Here is how main will print out the results:

```
   for (char let = 'A'; let <= 'Z'; let++)
   {
      // every 5 items, generate a newline
      if ( (let - 'A') % 5 == 0)
         System.out.println("");

      System.out.print( let + ": " + freq.getCount(let) + "      ");
   }
```

Copyright(c) 2006-2014 - Michael Loceff

Because I can design this class any way I want, I'm going to allow **getCount()** to take the chars 'A' through 'Z' and convert to the numbers 0 through 25 inside the class. Here is a sample run:

```
Enter a phrase or sentence:
If you give CTU McCarthy's name, they will eventually implicate the family

A: 5      B: 0      C: 4      D: 0      E: 7
F: 2      G: 1      H: 3      I: 6      J: 0
K: 0      L: 6      M: 4      N: 2      O: 1
P: 1      Q: 0      R: 1      S: 1      T: 6
U: 3      V: 2      W: 1      X: 0      Y: 5
Z: 0
```

8B.5.2 The Program Listing

Here is the entire listing in one place. Three classes. Try it out. Can you find an improvement that would enable **main()** to ignore completely the details of printing out the table? That would be a good modification for you to make.

```
import java.util.Scanner;

public class Sample
{
   public static void main (String[] args)
   {
      Scanner input = new Scanner(System.in);

      System.out.println("Enter a phrase or sentence: ");
      String userPhrase = input.nextLine();

      // create a CharacterCounter object for this phrase
      CharacterCounter freq = new CharacterCounter(userPhrase);

      // display whole table
      for (char let = 'A'; let <= 'Z'; let++)
      {
         // every 5 items, generate a newline
         if ( (let - 'A') % 5 == 0)
            System.out.println("");

         System.out.print( let + ": " + freq.getCount(let) + "      ");
      }
   }
}
```

Copyright(c) 2006-2014 - Michael Loceff

```
class CharacterCounter
{
   private Frequency letters;
   private String userString;

   public CharacterCounter(String str)
   {
      // instantiate a Frequency object of this instance
      letters = new Frequency(26);

      // convert string to uppercase for internal storage
      if (str != null && str.length() >=1 )
         userString = str.toUpperCase();
      else
         userString = "";
      countOccurrences();
   }

   private void countOccurrences()
   {
      char let;
      int k;

      // letters[] automatically initialized to all 0s
      // scan the string and increment as we go
      for (k = 0; k < userString.length(); k++)
      {
         let = userString.charAt(k);
         letters.increment( let - 'A' );
      }
   }

   public long getCount(char let)
   {
      char up_let;
      up_let = Character.toUpperCase(let);
      return letters.get(up_let - 'A');
   }
}
```

Copyright(c) 2006-2014 - Michael Loceff

```
class Frequency
{
   private long count[];
   private int size;
   private static int MAX_SIZE = 10000;

   public Frequency(int size)
   {
      if (size < 0 || size > MAX_SIZE)
         size = MAX_SIZE;

      this.size = size;
      count = new long[size];  // automatically sets to all 0s
   }

   public long get(int index)
   {
      if (index >= 0 && index < size)
         return count[index];
      else
         return -1;
   }
   public void increment(int index)
   {
      if (index >= 0 && index < size)
         count[index]++;
   }

   public void decrement(int index)
   {
      if (index >= 0 && index < size)
         if (count[index] > 0)
            count[index]--;
   }
}
```

Copyright(c) 2006-2014 - Michael Loceff

Section 6 - Allocating an Array of Objects

8B.6.1 Initializing an Array in the Declaration Statement

As we recently saw, we can not only assign values to an array of ***primitives*** or **Strings** in the declarations:

```
double[] myArray = {10.2, 56.9, -33, 12, 0, 2,
   4.8, 199.9, 73, -91.2};

String[] myArray = {"martin", "claudia", "sandra", "samuels",
   "terry", "jack", "clark", "palmer", "abraham", "Mike"};
```

but we can also assign user-defined objects in the array declaration statement:

```
Student[] myClass = { new Student("smith","fred", 95),
   new Student("bauer","jack",123),
   new Student("jacobs","carrie", 195),
   new Student("renquist","abe",148),
   new Student("3ackson","trevor", 108),
   new Student("perry","fred",225),
   new Student("loceff","fred", 44),
   new Student("stollings","pamela",452),
   new Student("charters","rodney", 295),
   new Student("cassar","john",321),
};
```

As you see, we are using the keyword **new** inside the braces for each individual **Student** in the array. This suggests that we have to allocate, or *instantiate*, an object for each element in an array of objects. This is true! It is not so obvious from the above example so let's see what happens when we do not initialize the array of objects directly in the array declaration statement. Where and how do we allocate the memory?

8B.6.2 The Program Listing

Often we don't know what to put into the array elements at the time of declaration. In these cases, we will ***instantiate the objects of the array in a loop*** when we do know or want to create the data. For example, what happens when we declare an array of **Students** called **cs1a** as follows:

```
final int NUM_STUDENTS = 5;
Student [] cs1a = new Student[NUM_STUDENTS];
```

Are we done allocating? Can we now start using the array in statements like:

```
Student.printArray("A Computer-Generated Roster", cs1a);
```

?

Copyright(c) 2006-2014 - Michael Loceff

No, we cannot. If we tried using this array we would get a run-time exception, citing null-pointers in the error. The array **cs1a** is not ready to be used after the above declaration, even though it contains a call to the **new** operator. What we have **new**-ed is not an ***array of objects***, but an ***array of references***. To use these references, we must follow the above with a loop that allocates each object, one-at-a-time:

```
   for (int k = 0; k < NUM_STUDENTS; k++)
      cs1a[k] = new Student(
                        "last_" + (char)('a'+k),
                        "first_" + (char)('a'+k),
                        k*100 );
```

Note that in each loop pass we ***new*** another object using the **Student** constructor with arguments. Here is the entire example along with the output (the Student class is repeated but unchanged):

```
import javax.swing.*;

public class Foothill
{
   public static void main (String[] args)
   {
      final int NUM_STUDENTS = 5;
      Student [] cs1a = new Student[NUM_STUDENTS];

      for (int k = 0; k < NUM_STUDENTS; k++)
         cs1a[k] = new Student(
            "last_"+ (char)('a'+k),
            "first_"+ (char)('a'+k), k
            *100 );

      StudentArrayUtilities.printArray("A Computer-Generated Roster", cs1a);
   }
}

class Student
{
   private String lastName;
   private String firstName;
   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;
```

Copyright(c) 2006-2014 - Michael Loceff

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

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

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

boolean setFirstName(String first)
{
   if ( !validString(first) )
      return false;
   firstName = first;
   return true;
}

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

Copyright(c) 2006-2014 - Michael Loceff

```
   // could be an instance method and, if so, would take one parameter
   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
         + ", " + firstName
         + " 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);
   }
```

Copyright(c) 2006-2014 - Michael Loceff

```
   // 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;
   }
}
```

That's it for this lesson.

Copyright(c) 2006-2014 - Michael Loceff