PRGM15: Dam Report

due: Monday 12/12/16, 12:00 noon (50 pts) – end of semester

Overview:

For this LAST program of the semester, you will one more time be working with arrays, except now it will be an array of OBJECTS – specifically, an array of ***Dam*** objects. Every element of any array is a like-typed data element, so here every array element **data[i]** will be a *Dam* object, and it can "do" anything that any *Dam* can do on its own.

This will also be an exercise in **"code reuse".** Real-world, efficient software development often depends on NOT reinventing the wheel, if you've previously gotten something similar working that can be repurposed. Here, you already have built certain parts of this program, now you will simply reuse them. Specifically:

- The ***while-switch*** **menuing structure** of PRGM11 will provide the overall program control
- Our old friend the ***Dam*** class will be used as a software component in this program
- Our ***Utils*** class will provide all low-level input utilities

In this program, you will do the following:

- **Read data** from a text file, and set up an array of Dam objects (***examples have been provided***)
- **Display** the used *Dams* of the array in two different ways: in a tabular column format, and showing all the *print()* details
- **Report** on the overall water system status or "health", by creating and aggregating an overall "Super Dam", and displaying it
- **Control** all of the above operations at the command line using a while/switch menu

I prefer that you NOT simply clone and alter the source code examples (other than your OWN menuing code). Instead, implement the needed code from scratch, but use the cited examples as your guides. Trust me, you will get much more out of the assignment this way, and it's often faster/less error prone to start with a clean slate, than to try and adapt code you haven't written yourself (I've seen this movie before..)

Java Objectives:

1) 1-D arrays, and array traversal using for-looping [CSLO-1]
2) Using an existing class (***Dam***) as a software component in a new program [CSLO-2]
3) Arrays of objects, manipulating indexed object array elements [CSLO-2]
4) Parsing strings using arrays [CSLO-1]
5) Initializing program data from a text file [CSLO-4]
6) Program decomposition into multiple utility methods [CSLO-3]
7) Using arrays as method arguments [CSLO-1]
8) Code reuse of existing programs, and adapting examples for new solutions [CSLO-4]
9) Output data formatting using format specifiers [CSLO-1]

Preliminaries:

1) As always, begin by FIRST reviewing all the current week's materials on **arrays**: lecture notes, videos, and source code examples. They should answer most, if not all, questions. If anything is still cloudy in your mind, please ask!

   In particular, focus on these topics:
   - In "**parsing**": reading line-by-line from a text file, parsing strings, and converting strings to objects. You will make use of these examples for two of your methods.
   - In "**scalars class**": the middle materials on arrays as method input/output parameters
   - The distinction between overall size and used size of an object array
   - You may also want to brush up on format specifiers for input/output

2) **Then, carefully study the extra provided example, *ObjectArrayMethodExamples.java*, which deals with an array of *CS12Date* objects, and several methods which operate upon it.**

   The general method structure of this example is **very similar** to that needed for this assignment. It contains several methods which all operate upon ONE common array of objects. Run the program a few times, and make sure you understand how it works.

3) **Just to put your mind at ease**, we are NOT going to implement a full-blown class which handles an object array. Instead, we'll do something a bit simpler, and just operate upon an object array using several static methods alongside the client *main()* method. See the **Structure** section diagram.

4) Make sure that you have a working ***Dam*** class available. At a minimum, the *Dam* part I portion of your class should be in good shape: constructors, accessors/mutators, and print(). **Review any past *Dam I* and *Dam II* rubric grading comments, and incorporate any needed updates.**

5) Look over the **program requirements**, **design notes**, **suggested implementation**, and **sample outputs**, to get a sense of what is being asked for in this assignment.

6) Begin by copying your old **HybridLogicFL.java** class into **DamReportFL.java**.

   Then, "gut" the program by removing all the *if-else* and *for* looping inside the switch cases. Leave the menu and the *while* and *switch* structures intact, but empty. Make sure your (empty) framework still compiles and runs! You might want to test it with some starting placeholder *println()* statements in each case. Don't forget to update the comment block accordingly.

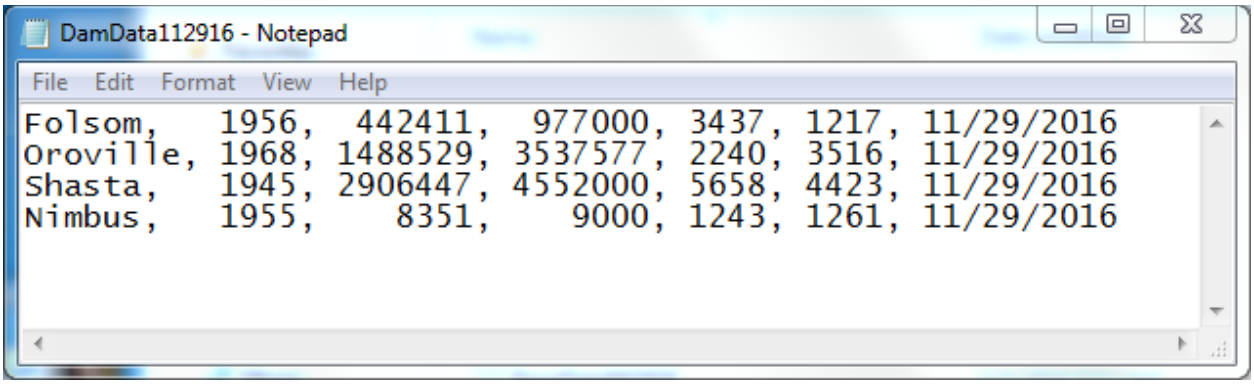
7) Continue using your ***Utils*** class for any data input needs.

Requirements:

1. **Purpose:**
    - **a.** The purpose of this program is set up an array of ***Dam*** objects, initialized from a text file, together with several local static methods which operate upon it.
    - **b.** All available methods should be controlled from one-line method calls from a *while-switch* menu.
2. **Application structure:** Your program should consist of these 3 files:
    - **a.** A new class ***DamReportFL***, with a *main()* client method and the 5 methods listed in the provided API (6 methods total).
    - **b.** Your existing ***Dam*** class. No mods should be needed, but make any needed rubric fixes.
    - **c.** Your existing ***Utils*** class, to be used for *char* and *String* inputs only.
3. **Client structure**: Your client program should contain the following elements:
    a. An array of ***Dam*** objects, size this to 50 (not all will be used).
    b. An integer count of actual dams being used.
    c. A menuing structure (*while* + *switch*), with 4 lettered menu options corresponding to API methods 2 thru 5, with each *switch* case calling ONE of the API methods.
    d. The menu must be case-insensitive, and must loop until some terminate option is read.
4. **API**:
    a. To expedite grading, the API naming shown in the **Structure** and **Design** sections must be implemented specifically.
    b. See the **Design** section for descriptions of all 5 needed methods and their operation.
5. **Inputs**:
    a. Menu switch options must be read using ***Utils readChar().***
    b. The filename for data input must be read using ***Utils readString().***
    c. A ***Dam*** data file is provided as **DamData112916.txt.** It has similar data as for Dam II, although it reflects later season water conditions**. For read, you can assume this file exists.**
    d. Data is in API instance variable order (name, year, storage, capacity, inflow, outflow, date).
    e. Other data files (longer OR shorter) must also work and will be used to test program.
6. **Outputs**:
    - **a.** When reading in data from file, echo to user how many lines were read and from what file.
    b. When displaying dam summaries, print data in fixed-width, tabular format using the ***Dam*** accessors for each piece of data. ***Outputs should resemble the sample outputs provided***.
        i. Print data column headers
        ii. Left-justify name, right-justify all other fields. **Use format specifiers + printf().**
    c. When displaying dam details, simply print each dam array element using its own *print().*
    d. When displaying overall water status:
        i. Sum all water data (4 doubles), using one or more *for* loops.
        ii. Use the current year for the year, date of any individual dam (all are the same).
        iii. Use some made-up summary title for the dam name.
        iv. Assemble all data into one new ***Dam*** object "Super Dam", and *print()* it.

Structure:

Here is the Dam information you will be working with. This data is similar to that used in in Dam II, except updated for more current water conditions. **This is NOT the same data you used before**:



We want to read all this data into our program, and turn it into an array of ***Dam*** objects, so that we can display it and perform computations with it. **When reading this file, you may assume that it exists in the local directory (no error checking required).**

Each line of the file will correspond to ONE ***Dam*** object in the array. All data is in published ***Dam*** API order. You will need to parse each line using the ***String split()*** method. Notice that you will need to perform a secondary ***split()*** upon the date. **See the provided examples for reading data from a file, into objects in an object array.**

Once all this data has been read into an array of Dams, each i-th array element is:

dams[i] ← → one Dam object

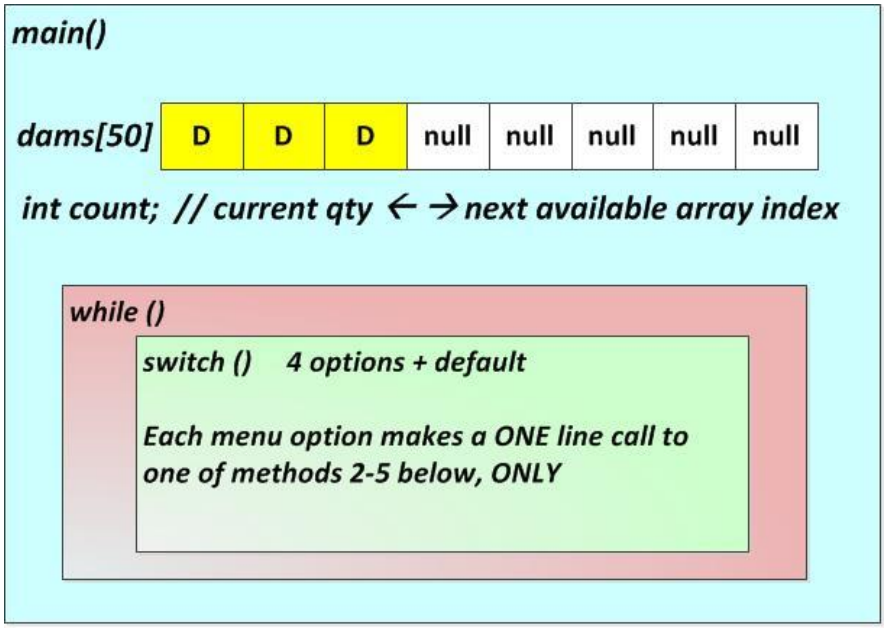
and we can "do" with any array element anything we could do with any other ***Dam*** object, using **dot notation** upon the array element. For example:

dams[i].print("I am a Dam"); // printing the i-th Dam

theName = dams[i].getName(); // getting the name of the i-th Dam

The needed structure of the new ***DamReport*** class is shown below.

- There is one *main()* method, containing the object array and an "elements used" count. Here, the count=3, which also doubles as the index of the next available array element.
- The *main()* method also contains a *while/switch* menu of 4 options. **Each switch case option contains a one-line call ONLY to one of methods 2-5**. Method 1 is called by method 2.
- The Dam array name, and possibly also the count, are passed as inputs to the various methods. Methods 2-5 all operate upon the SAME array of dams.



convertString2Dam()
method to convert an input String to a Dam object

readDataFromFile()
method to read text file and insert Dams into array

printDamSummaries()
method to print all used Dams in a tabular format

printDamDetails()
method to print all used Dams using their own print()

showWaterStatus()
method to loop thru used Dams, sum all water data, create new "super Dam" and print() it

See the Design section which follows, for the exact interfaces and needed details of each of the above 5 methods.

Design:

Here is the required API. These 5 methods must be named exactly as indicated. **All methods should be declared as *private static*.**

Methods 2-5 are called from one of the 4 *main() switch* cases. Method 1 is called internally by method 2.

Return type	Method and interface	Comments
DamFL	***convertString2Dam****(String data)*	• Parses an input String, assembles and returns a Dam object. • Implements a String-to-Dam conversion. • *See **createPerson()** of **SplitPerson.java** sample code for the example of how this should work.*
int	**readDataFromFile**(DamFL [ ] dams)	• Prompts user for a filename, uses ***File*** and ***Scanner*** classes to read line-by-line from this text file, using a *while* loop. Data is in ***Dam*** API order in the file. • Before reading any data, initialize the ENTIRE array by nulling all elements. So each read wipes out any prior data. • For each line read, convert it to a ***Dam*** object using ***convertString2Dam(),*** and insert ***Dam*** into the provided ***Dams*** array, then increment the count. • **Return count of *Dams* read to the caller.** • Print to the screen the number of dams read and from which file. • See ***main()*** of ***SplitPersonFileRead.java*** sample code for the example of how this should work.
void	**printDamSummaries**(DamFL [ ] dams, int count)	• Given a client ***Dams*** array and a count of used elements, prints all Dams as one-line records, in tabular format. See provided sample output. • Loop over all used ***Dam*** array elements using provided count. • Print all fields of each Dam using its accessor methods. • Use **format specifiers** to format each line: no decimal places on water data, left-justify dam name, right-justify all numeric data. • Print a data header as shown in sample output. • If no dams exist, display an error message to user.
void	**printDamDetails**(DamFL [ ] dams, int count)	• Given a client ***Dams*** array and a count of used elements, prints all details for all ***Dams***. • Loop over all used ***Dam*** array elements using provided count. • Print all details for each ***Dam***, using its ***print()*** method. • Use each dam's name as its print title. • If no dams exist, display an error message to user.
void	**showWaterStatus**(DamFL [ ] dams, int count)	• Given a client ***Dams*** array and a count of used elements, loops over all used dams and sums up 4 water-related quantities, and using them to create a new "Super Dam". • For name, use name as shown in sample output. • For year, simply use current year (**DO NOT HARDCODE, use a *Utils* method**) • For date, use the date of any ONE dam (all are same). • Do NOT add dam to array, just print it using ***print().*** • If no dams exist, display an error message to user.

Suggested Implementation:

This should not be a terribly difficult program to write, IF you set up a logical framework for it FIRST, and then gradually flesh out the details: ***"divide and conquer"*!**

For good results and minimal stress, I suggest using the following development "roadmap":

1. Check your **recent grading rubrics**, and incorporate any needed fixes to your ***Dam*** or ***Utils***.
2. First, **set up a menu structure** with the needed letter options. This is most easily done by copying over your ***HybridLogic*** program ("code reuse"), and "gutting" the switch options content, then setting up the needed lettered cases. Before proceeding, **add some placeholder print statements to each switch case**, to make sure your existing structure still works.
3. Next, **set up empty method stubs at the end of your file** for the needed 5 methods. For now, simply make them 0-input, 0-output method stubs. Worry about the correct method interface later. I suggest you put in some print statements for now as placeholder code.
4. Next, **incorporate the needed 4 method calls into your *switch* case options**. Test and make sure the correct placeholder messages are displayed, and that your menu looping is still working as it should. Make sure case-insensitive handling is working.
5. **Add the needed Dam array and count variable** to your ***main()*** method. Set the count to 0, and simply declare and instantiate the array to size 50, which is way more than we'll need (intentionally).
6. Next, **implement a STARTER version of readDataFromFile().** Don't worry about reading from a file yet, simply create 4-5 default Dams and insert them into the array for now. Up the count for each!
7. **Test your initial array** by implementing **printDamDetails().** It simply loops thru the used portion of the array, and calls each ***Dam***'s ***print()*** to display its details. Test both these first two options from your menu.
8. Next**, implement printDamSummaries().** Start out by printing each ***Dam*** in a loop using its ***toString().*** Then, replace this by using the accessor methods to pull out the 7 instance variables, and printing them in tabular fashion using a format specifier. Test this option from your menu.
9. **Implement convertString2Dam(). Follow the cited example code carefully**, and this should be relatively straightforward. Pay careful attention to the needed indices in the two *String* arrays.
10. **Revise your starter readDataFromFile()** so that it now reads from a prompted filename. Again, if you **follow the cited example closely**, this should be straightforward. Incorporate the above method to convert a *String* into a ***Dam***. Test the file reading from your menu.
11. Then, **implement showWaterStatus().** Loop over all the used ***Dams***, tally up all the water quantities, and use these as indicated to create a "super dam" representing the total water system, and ***print()*** it out (do NOT add it to the array!). Test this option from your menu.
12. Finally, TEST all options in series from your menu: read from file, import data, display data two ways, and print the summary information. Make sure all options work, including the Quit option.
13. **Check your results numerically versus the published sample outputs on the following pages!**
14. Make up some new file data, with both more and fewer dams, and make sure the data is handled and displayed correctly. Each new read of data must OVERWRITE any prior data.

Testing:

In testing your code, make sure to check the following things:

- **Good data**: must echo how many lines read and from what file.
- **No data**: must inform the user if there are no dams in the array (such as before a file gets read)
- **Summary formatting**: fixed-width columns, numeric data right-justified, header labels.
- **Print formatting**: make sure any rubric comments concerning print() formatting are addressed.
- **Numerical correctness**: your outputs should numerically MATCH the provided sample outputs, given the same input data file.
- **Overall status**: some overall name, current year, use of one common measurement date.
- **Different data files**: files of different sizes must each work, and each read must clear any prior data. Test your code with cloned data files which are longer OR shorter.

Sample output:

Your resulting outputs should look something like this. **Note that this is NOT the same data as for Dam II. You should be able to duplicate these results!**

```
----jGRASP exec: java DamReportRL

DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > s
ERROR: no Dams currently exist! Must import from file.

DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > d
ERROR: no Dams currently exist! Must import from file.

DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > w
ERROR: no Dams currently exist! Must import from file.

DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > r
Enter text file containing Dam data: DamData112916.txt
        4 dams read from file: DamData112916.txt
```

```
DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > s
```

Name	Year	Storage	Capacity	Inflow	Outflow	Date
Folsom	1956	442411	977000	3437	1217	11/29/2016
Oroville	1968	1488529	3537577	2240	3516	11/29/2016
Shasta	1945	2906447	4552000	5658	4423	11/29/2016
Nimbus	1955	8351	9000	1243	1261	11/29/2016

```
DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > d
======
Folsom
======
name:                     Folsom
year opened:              1956
age [yrs]:                60
data as of:               11/29/2016
storage [acre-ft]:        442,411
capacity [acre-ft]:       977,000
inflow [cu-ft/s]:         3,437
outflow [cu-ft/s]:        1,217
% full:                   45.3 %
status:                   filling
days until dam event:     121
date of dam event:        3/30/2017

========
Oroville
========
name:                     Oroville
year opened:              1968
age [yrs]:                48
data as of:               11/29/2016
storage [acre-ft]:        1,488,529
capacity [acre-ft]:       3,537,577
inflow [cu-ft/s]:         2,240
outflow [cu-ft/s]:        3,516
% full:                   42.1 %
status:                   emptying
days until dam event:     588
date of dam event:        7/10/2018
```

```
======
Shasta
======
name:                       Shasta
year opened:                1945
age [yrs]:                  71
data as of:                 11/29/2016
storage [acre-ft]:          2,906,447
capacity [acre-ft]:         4,552,000
inflow [cu-ft/s]:           5,658
outflow [cu-ft/s]:          4,423
% full:                     63.8 %
status:                     filling
days until dam event:       671
date of dam event:          10/1/2018

======
Nimbus
======
name:                       Nimbus
year opened:                1955
age [yrs]:                  61
data as of:                 11/29/2016
storage [acre-ft]:          8,351
capacity [acre-ft]:         9,000
inflow [cu-ft/s]:           1,243
outflow [cu-ft/s]:          1,261
% full:                     92.8 %
status:                     emptying
days until dam event:       233
date of dam event:          7/20/2017
```

```
DAM OPTIONS:
Read data from file   [R]
Print dam summaries   [S]
Print dam details     [D]
Overall water status  [W]
Quit                  [Q]
Enter option > w

name:                       OVERALL WATER HEALTH
year opened:                2016
age [yrs]:                  0
data as of:                 11/29/2016
storage [acre-ft]:          4,845,738
capacity [acre-ft]:         9,075,577
inflow [cu-ft/s]:           12,578
outflow [cu-ft/s]:          10,417
% full:                     53.4 %
status:                     filling
days until dam event:       986
date of dam event:          8/12/2019

DAM OPTIONS:
Read data from file   [R]
Print dam summaries   [S]
Print dam details     [D]
Overall water status  [W]
Quit                  [Q]
Enter option > h
ERROR: Unrecognized option h, please try again!

DAM OPTIONS:
Read data from file   [R]
Print dam summaries   [S]
Print dam details     [D]
Overall water status  [W]
Quit                  [Q]
Enter option > q
Exiting upon user request, goodbye!

 ----jGRASP: operation complete.
```

For testing purposes, you should also be able to read different-sized files. Using the provided data, copy and edit it into 2-dam and 6-dams versions, and try reading them also. Each time, your code should clear out the object array and start fresh:

```
----jGRASP exec: java DamReportRL

DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > r
Enter text file containing Dam data: Dams2.txt
        2 dams read from file: Dams2.txt

DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > s

Name        Year   Storage  Capacity  Inflow Outflow        Date
Folsom      1956    442411    977000    3437    1217  11/29/2016
Oroville    1968   1488529   3537577    2240    3516  11/29/2016

DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > r
Enter text file containing Dam data: Dams6.txt
        6 dams read from file: Dams6.txt

DAM OPTIONS:
Read data from file  [R]
Print dam summaries  [S]
Print dam details    [D]
Overall water status [W]
Quit                 [Q]
Enter option > s

Name        Year   Storage  Capacity  Inflow Outflow        Date
Folsom      1956    442411    977000    3437    1217  11/29/2016
Oroville    1968   1488529   3537577    2240    3516  11/29/2016
Shasta      1945   2906447   4552000    5658    4423  11/29/2016
Nimbus      1955      8351      9000    1243    1261  11/29/2016
Shasta      1945   2906447   4552000    5658    4423  11/29/2016
Nimbus      1955      8351      9000    1243    1261  11/29/2016
```

Submitting the assignment:

Turn in the **3** text Java files **DamReportFL.java, DamFL.java,** and **UtilsFL.java.** You don't need to turn in **CS12Date.java** or the **data file**, I already have those.

Submit all 3 Java source files in Canvas, under this assignment.

Make sure your code adheres to all the usual coding conventions described in the course Coding Standard. **Check your program against the rubric below before submitting it!**

Grading:

See the rubric associated with this assignment in Canvas, or below.

	PRGM15: TOTAL POINTS POSSIBLE:	**50**
1.00	**Program Elements/Structure**	**23**
1.01	Correctly-sized Dam array and integer count	1
1.02	(cloned) while loop/switch structure, with all 5 needed menu options, plus default	2
1.03	Each of 4 switch options are only 1-line method calls	1
1.04	5 static helper methods, adherence to API naming, correct interfaces	2
1.05	The 3 print and overall status methods check for no dams present + print error message	2
1.06	String to Dam: 2-way string splits, assembly into Dam object	3
1.07	Read from file: nulling of array, reading of lines, array inserts, count returned AND printed	3
1.08	Print summary: format specifiers, alignments, all 7 instance vars printed using accessors	3
1.09	Print details: looping, titled dams, print()	3
1.10	Water status: summing of water quantities, name/date/year, print()	3
2.00	**Program Execution and Testing**	**20**
2.01	Compiles and runs as submitted	2
2.02	All menu options implemented (including default), loops properly until Quit	2
2.03	Error message printed if no dams present, for 3 print and overall status methods	2
2.04	All lines of data file read, no old data, filename and # of lines reported	2
2.05	Summary print: all data, column headers, column format, no decimals, proper alignments	2
2.06	Detail print: header title, all data, print() for each one	2
2.07	Water status: summing is performed, "super dam" is properly created and print()'ed	2
2.08	Numerical correctness of "super dam", matches sample output	2
2.09	Numerical correctness+formatting of individual dams (reflects Dam II updates/fixes)	2
2.10	Other data files (shorter and/or longer) also read correctly	2
3.00	**Program Style**	**7**
3.01	General adherence to coding standard	4
3.02	Adequate commenting of all program logic, one-liners preceding all methods	3
4.00	**Late Deduction**	**0**
4.01	DUE AS ASSIGNED, I will be a little flexible, but also need to get grades posted	0