Deliverables

Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. In order to avoid a late penalty for the project, you must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code assignment. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your TA before the deadline. Test files are not required for this project. If submitted, you will be able to see your code coverage, but this will not be counted as part of your grade.

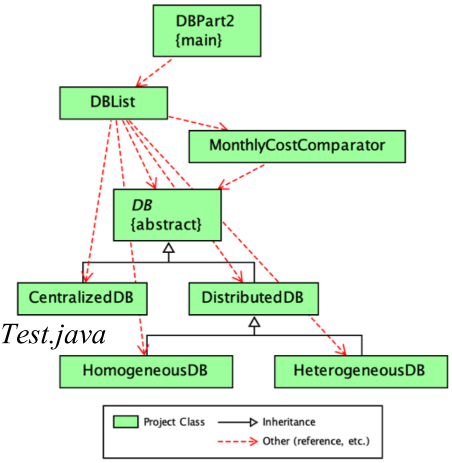
Files to submit to Web-CAT (*test files are optional*):

From Database – Part 1

- DB.java
- CentralizedDB.java, *CentralizedDBTest.java*
- DistributedDB.java, *DistributedDBTest.java*
- HomogeneousDB.java, *HomogeneousDBTest.java*
- HeterogeneousDB.java *HeterogeneousDBTest.java*

New in Database – Part 2

- MonthlyCostComparator.java, *MonthlyCostComparatorTest.java*
- DBList.java, *DBListTest.java*
- DBPart2.java, *DBPart2Test.java*



Recommendations

You should create new folder for Part 2 and copy your relevant Part 1 source and optional test files to it. You should create a jGRASP project with these files in it, and then add the new source and optional test files as they are created.

Specifications – Use arrays in this project; ArrayLists are not allowed!

Overview: This project is the second of three that will involve the monthly cost and reporting for database systems. In Part 1 you developed Java classes that represent categories of database including centralized database and distributed database (both homogeneous and heterogeneous). In Part 2, you will implement three additional classes: (1) MonthlyCostComparator that implements the Comparator interface for DB, (2) DBList that represents a list of database objects and includes several specialized methods, and (3) DBPart2 which contains the main method for the program. Note that the main method in DBPart2 should create a DBList object and then call the readFile method on the DBList object, which will add DB objects to the list as the data is read in from a file. You can use DBPart2 in conjunction with interactions by running the program in a jGRASP canvas (or debugger with a breakpoint) and single stepping until the variables of interest are created. You can then enter interactions in the usual way. In addition to the source files, you may create an optional JUnit test file for each class and write one or more test methods to ensure the classes and methods meet the specifications. You should create a jGRASP project upfront and then add the new source and optional test files as they are created. All of your files should be in a single folder.

- **DB.java**

  Requirements and Design: In addition to the specifications in Part 1, the DB class should implement the Comparable interface for DB, which means the following method must be implemented in DB.

  - compareTo: Takes a DB object as a parameter and returns an int indicating the results of comparing the two DB objects based on their respective name fields ignoring case.

- **CentralizedDB, DistributedDB, HomogeneousDB, and HeterogeneousDB**

  Requirements and Design: No changes from the specifications in Part 1.

- **DBList.java**

  Requirements: The DBList class provides methods for reading in the data file and generating reports.

  Design: The DBList class has fields, a constructor, and methods as outlined below.

  (1) **Fields:** (1) An array of DB objects and (2) an array of String elements to hold invalid records read from the data file. [The second array will be used in Part 3.] Note that there are no fields for the number elements in each array. In this project, the size of the array should be the same as the number of DB objects in the array. These two fields should be private.

  (2) **Constructor:** The constructor has no parameters and initializes the DB array and String array in the fields to arrays of length 0.

  (3) **Methods:** Usually a class provides methods to access and modify each of its instance variables (i.e., getters and setters) along with any other required methods. The methods for DBList are described below.

  - getDBArray returns an array of type DB representing the DB array field.
  - getInvalidRecordsArray returns an array of type String representing the invalid records array field.
  - addDB has no return value, accepts a DB object, increases the capacity of the DB array by one, and adds the DB object in the last position of the DB array. See Hints on last page.
  - addInvalidRecord has no return value, accepts a String, increases the capacity of the invalidRecords array by one, and adds the String in the last position of the invalidRecords array. This method will be used in the next project, but it still needs to be tested in this project. See Hints on last page.
  - readFile has no return value, accepts the data file name as a String, and throws FileNotFoundException. This method creates a Scanner object to read in the file one line at a time. When a line is read, a separate Scanner object on the line should be created to read the values in that line. The data in each line is separated by a comma so the

delimiter should be set to comma by invoking the `useDelimiter(",")` method on the Scanner object for the line. For each line read in, the appropriate DB object is created and added to the DB array field, or if not a valid category code, the line should be ignored. The data file has comma-delimited text records as follows: category, name, base storage cost, followed by one or more fields specific to the category. Remember, CentralizedDB, DistributedDB, HomogeneousDB, and HeterogeneousDB objects are all DB objects. The category codes are C for CentralizedDB, D for DistributedDB, H for HomogeneousDB, and E for HeterogeneousDB. Any other category code is invalid. Below are examples data records:

```
C,Database One,1200.0,5.0,1500
D,Database Two,2000.0,7.5,100,12.0
H,Database Three,2000.0,7.5,100,14.0
Z,Database Zero,2000.0,7.5,100,16.0
E,Database Four,2000.0,7.5,100,14.0
```

- `generateReport` processes the DB array using the original order from the file to produce the Monthly Database Report and then returns the report as String. See example result in output for DBPart2 beginning on page 5.
- `generateReportByName` sorts the DB array by its natural ordering, and processes the DB array to produce the Monthly Database Report (by Name), then returns the report as a String. See example result in output for DBPart2 beginning on page 5.
- `generateReportByMonthlyCost` sorts the DB array by monthly cost, and processes the DB array to produce the Monthly Database Report (by Monthly Cost) and then returns the report as String. See example result in output for DBPart2 beginning on page 5.

Code and Test: For each of the three generateReport methods above, the return value does not begin with \n character, then after the title lines, each DB toString return value has \n\n appended to it. Hence, the return value for each of the three generateReport methods above ends \n\n.

See examples of file reading and sorting (using Arrays.sort) in the class notes. The natural sorting order is based on a DB object's name and is determined by the compareTo method when the Comparable interface is implemented. The following call to Arrays.sort can be used to sort the DB array in `generateReportByName` above.

```
      Arrays.sort(getDBArray());
```

The sorting order based on a DB object's monthly cost is determined by the MonthlyCostComparator class which implements the Comparator interface (described below).

```
  Arrays.sort(getDBArray(), new MonthlyCostComparator());
```

If you have an optional test file with test methods for the generate reports methods above, you may want to use the following assertion to avoid having to match the return result exactly (where the expected_result is part of what you think it should contain and the actual_result is the result of the method call.

```
      Assert.assertTrue(actual_result.contains(expected_result));
```

- **MonthlyCostComparator.java**

  Requirements and Design: The MonthlyCostComparator class implements the Comparator interface for DB objects. Hence, it implements the method `compare(DB d1, DB d2)` that defines the ordering from **highest to lowest** based on the database monthly cost. See examples in class notes.

- **DBPart2.java**

  Requirements: The DBPart2 class contains the main method for running the program.

  Design: The DBPart2 class is the driver class and has a main method described below.

  - `main` accepts a file name as a command line argument, creates a DBList object, and then invokes its methods to read the file and process the database records and then to generate and print the three reports as shown in the example output beginning on page 5. If no command line argument is provided, the program should indicate this and end as shown in the first example output on page 5. An example data file can be downloaded from the assignment page in Canvas.

  Code and Test: When printing the return values of the generateReport methods, use `print` rather than `println` to achieve proper spacing.

  If you have an *optional* test file for the DBPart2 class, you should have at least two test methods for the main method. One test method should invoke DBPart2.main(args) where args is an empty String array, and the other test method should invoke DBPart2.main(args) where args[0] is the String representing the data file name. Depending on how you implemented the main method, these two methods should cover the code in main. As for the assertion in the test method, since `COST_FACTOR` is a public class variable in DistributedDB, you could assert that `DistributedDB.COST_FACTOR` equals `1.1` in each test method.

  In the first test method, you can invoke main with no command line argument as follows:

```
// If you are checking for args.length == 0
// in DBPart2, the following should exercise
// the code for true.
String[] args1 = {};  // an empty String[]
DBPart2.main(args1);
```

In the second test method, you can invoke main as follows with the file name as the first (and only) command line argument:

```
String[] args2 = {"database_data_1.csv"};
// args2[0] is the file name
DBPart2.main(args2);
```

If Web-CAT complains the default constructor for `DBPart2` has not been covered, you may want to include the following line of code in one of your test methods to exercise the constructor.

```
// to exercise the default constructor
DBPart2 app = new DBPart2();
```

Notes:

1. Passing in command line arguments in jGRASP – On the top menu, click “Build” then turn on “Run Arguments” by clicking the associated checkbox. Now you can enter the arguments (e.g., the filename) in the Run Arguments text box at the top of the edit window containing the main method. Finally, run or debug the program in the usual way.

2. To run the program with no command line argument, either delete the text entered above. Alternatively, click “Build” then turn off “Run Arguments” by clicking the associated checkbox. Then run or debug the program in the usual way.

3. You can also test your program using your own data files.

Example Output when file name is missing as command line argument

```
 ----jGRASP exec: java DBPart2
File name expected as command line argument.
Program ending.

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

Example Output for *database_data_1.csv*

```
 ----jGRASP exec: java DBPart2 database_data_1.csv
-------------------------------
Monthly Database Report
-------------------------------
Database One (class CentralizedDB) Monthly Cost: $2,700.00
Storage: 5.000 TB
Base Cost: $1,200.00
License: $1,500.00

Database Two (class DistributedDB) Monthly Cost: $3,320.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $12.00
User Cost: $1,200.00
Cost Factor: 1.1
```

```
Database Three (class HomogeneousDB) Monthly Cost: $3,680.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.2

Database Four (class HeterogeneousDB) Monthly Cost: $3,820.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.3

-----------------------------------------
Monthly Database Report (by Name)
-----------------------------------------
Database Four (class HeterogeneousDB) Monthly Cost: $3,820.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.3

Database One (class CentralizedDB) Monthly Cost: $2,700.00
Storage: 5.000 TB
Base Cost: $1,200.00
License: $1,500.00

Database Three (class HomogeneousDB) Monthly Cost: $3,680.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.2

Database Two (class DistributedDB) Monthly Cost: $3,320.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $12.00
User Cost: $1,200.00
Cost Factor: 1.1

-------------------------------------------------
Monthly Database Report (by Monthly Cost)
-------------------------------------------------
Database Four (class HeterogeneousDB) Monthly Cost: $3,820.00
Storage: 7.500 TB
Base Cost: $2,000.00
```

```
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.3

Database Three (class HomogeneousDB) Monthly Cost: $3,680.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $14.00
User Cost: $1,400.00
Cost Factor: 1.2

Database Two (class DistributedDB) Monthly Cost: $3,320.00
Storage: 7.500 TB
Base Cost: $2,000.00
Number of Users: 100
Cost per User: $12.00
User Cost: $1,200.00
Cost Factor: 1.1

Database One (class CentralizedDB) Monthly Cost: $2,700.00
Storage: 5.000 TB
Base Cost: $1,200.00
License: $1,500.00


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

Hints

1. Adding an element to a full array in your **addDB** and **addInvalidRecord** methods – Consider the example below where MyType[] myArray is an instance field and addElement is an instance method that adds newElement to myArray, which is full. Since the length of an array cannot be changed after it has been created, myArray must be replaced with one that has a length of myArray.length + 1 and then elements from the original array must be copied to the new array. This copy operation could be done using a loop. However, Java.util.Arrays provides a copyOf method, which creates the new array and performs the copy in a single statement as shown in the first statement in the method below. The second statement adds newElement as the last element in the array.

   ```
   public void addElement(MyType newElement) {
      myArray = Arrays.copyOf(myArray, myArray.length + 1);
      myArray[myArray.length - 1] = newElement;
   }
   ```

2. The advantage to keeping the array full is that it allows the use of for-each loops with the array.

   ```
   for (MyType mt : myArray)
   {
      // do something with each mt
   }
   ```

3. In the readFile method, if you use a switch statement to determine the category, you should use type char for the switch expression rather than String; that is, each of the case labels should be of type char (e.g., `case 'D':` rather than type String (e.g., `case "D":`). When the switch type is String, the code coverage tool used by Web-CAT fails to detect that the default case is covered. If category is the reference to the String that contains the category code, then the following statement returns the category code as type char.

   ```
   category.charAt(0)
   ```