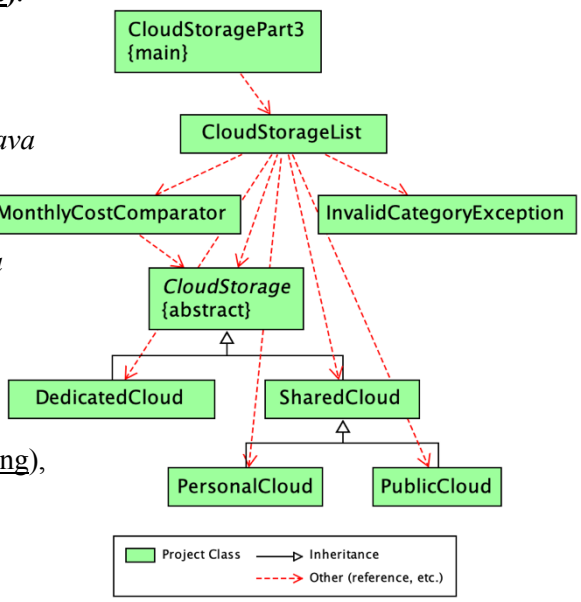
Project: Cloud Storage – Part 3

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 Cloud Storage – Part 1

- CloudStorage.java
- DedicatedCloud.java, *DedicatedCloudTest.java*
- SharedCloud.java, *SharedCloudTest.java*
- PublicCloud.java, *PublicCloudTest.java*
- PersonalCloud.java, *PersonalCloudTest.java*

From in Cloud Storage – Part 2

- MonthlyCostComparator.java, *MonthlyCostComparatorTest.java*
- CloudStorageList.java (add exception handling), *CloudStorageListTest.java*

New in Cloud Storage – Part 3

- CloudStoragePart3.java (with exception handling), *CloudStoragePart3Test.java*
- InvalidCategoryException

Recommendations

You should create new folder for Part 3 and copy your relevant Part 2 source and optional test files to it. You should create a jGRASP project and add the new source and any new optional test files as they are created.

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

Overview: Cloud Storage – Part 3 is the third of a three-part software project that involves the monthly cost and reporting for cloud storage. The completed class hierarchy is shown in the UML class diagram above. Part 3 of the project focuses on handling exceptions that are thrown as a result of erroneous input from the command line or the data file. In the CloudStoragePart3 class, the main method, which reads in the file name as a command line argument, will need to handle a FileNotFoundException that may result

from attempting to open the file (e.g., if the file does not exist). Also, the readFile method in CloudStorageList will need to handle exceptions that occur while processing the data file, including a new exception called InvalidCategoryException.

- **CloudStorage, DedicatedCloud, SharedCloud, PublicCloud, PersonalCloud, and MonthlyCostComparator**

  Requirements and Design: There are no changes to these classes from Part 2.

- **CloudStorageList.java**

  Requirements: The CloudStorageList class provides methods for reading in the data file and generating the reports. The readFile method should redesigned to handle exceptions in the data. Reading a "good" line of data results in a new element being added the CloudStorage array, and reading a "bad" line of data results in the line + an exception message being added the to the invalid records String array. A new report method produces the Invalid Records Report.

  Design: The readFile method from Part 2 should be redesigned to handle exceptions. The CloudStorageList class has fields, a constructor, and methods as outlined below.

  (1) **Fields:** no change from Part 2.

  (2) **Constructor:** no change from Part 2.

  (3) **Methods:** The readFile method needs to be reworked and the generateInvalidRecordsReport method needs to be added. See Part 2 for the full description of other methods in this class.

  - `readFile` has no return value, accepts the data file name as a String, and throws FileNotFoundException. If a FileNotFoundException occurs when attempting to open the data file, it should be ignored in this method so that it can be handled in the calling method (i.e., main). If a line from the file is processed successfully, a CloudStorage object of the appropriate category (subclass) is added to the CloudStorage array in the class. However, when an exception occurs as a result from erroneous data in a line read from the file, it should be caught and handled as follows. The line should be concatenated with a newline and the exception message and then the resulting String should be added to the invalid records String array in the class. The three exceptions that should be caught in this method are (1) InvalidCategoryException (described below), (2) NumberFormatException, and (3) NoSuchElementException. Note that the InvalidCategoryException must be explicitly thrown by your code if the category is not D, S, C, or P (*hint: the default case*). The NumberFormatException will be thrown automatically if the item scanned in the line from the file is not a double when Double.parseDouble expects it to be a double. The NoSuchElementException will be thrown automatically if the item scanned does not exist (i.e., data is missing). For examples, see the output below for the Invalid Records Report.

  - `generateInvalidRecordsReport` processes the invalid records array to produce the Invalid Records Report and then returns the report as String. See the example result near the end of the output for CloudStoragePart3 that begins on page 4 and ends on page 6.

Code and Test: See examples of exception handling in the text and the class notes. In the catch blocks for the NumberFormatException and NoSuchElementException, the invalid line should be concatenated with a newline and the exception message, and the resulting String should be added to the invalidRecords String array. Note that for the NoSuchElementException, `": For missing input data"` will need to be concatenated to the end of the toString value of the NoSuchElementException to form the complete message.

Download `cloud_storage_data_2_exceptions.csv` from the assignment page in Canvas to test your program. Your optional JUnit test methods should force the exceptions described above to thrown and caught. Since the readFile method will propagate the FileNotFoundException if the file is not found when the Scanner is created to read the file, test method could catch this exception to check that it was thrown. Any other test method involving the readFile method must have the throws FileNotFoundException clause.

InvalidCategoryException.java

Requirements and Design: The InvalidCategoryException class defines a new subclass of the Exception class. The constructor accepts a String `categoryIn` representing the invalid category, then invokes the super constructor with the message:

```
"For category: " + categoryIn
```

See **examples** of creating user defined exceptions in text and class notes.

CloudStoragePart3.java

Requirements: The CloudStoragePart3 class contains the main method for running the program. In addition to the specifications in Part 2, the main method should be modified as indicated below.

Design: The CloudStoragePart3 class is the driver class and has a main method described below.

- `main` accepts a file name as a command line argument, then within a try block, creates a CloudStorageList object, and then invokes its methods to (1) read the file and process the wireless network records and (2) to generate and print the four reports as shown in the third run in example output beginning on page 4. If no command line argument is provided, the program should indicate this and end as shown in the first run in the example output on page 4. If an FileNotFoundException is thrown in the readFile method in the CloudStorageList class, it should be caught in the catch block of the try statement in main. The catch block should print a message ("*** Attempted to read file: " along with the exception's message). For example, if the user entered "nofile.csv" as the command line argument and this file does not exit, then the Run I/O in jGRASP would look like the second run in the example output beginning on page 4). Note that since the main method is catching FileNotFoundException, it no longer needs the throws clause in its declaration.

Code and Test: See examples of exception handling in the text and the class notes. Download `cloud_storage_data_2_exceptions.csv` from the assignment page in Canvas to test your program. One of your optional JUnit test methods should call your main method with no argument (i.e., an empty String array). Another should call your main method with an argument that is not a valid file name to ensure that your catch block is covered. Finally, a third should call your main method with an argument that is the file name above. See "Code and Test" for CloudStoragePart2 in Part 2 to see how to invoke your main method.

Example Output

Three separate runs are shown below: (1) one with no command line argument, (2) one with an invalid file name as command line argument, and (3) one with valid file name as command line argument.

```
 ----jGRASP exec: java CloudStoragePart3
File name expected as command line argument.
Program ending.

 ----jGRASP: operation complete.

 ----jGRASP exec: java CloudStoragePart3 nofile.csv
 *** Attempted to read file: nofile.csv (No such file or directory)

 ----jGRASP: operation complete.

 ----jGRASP exec: java CloudStoragePart3 cloud_storage_data_2_exceptions.csv
------------------------------
Monthly Cloud Storage Report
------------------------------
Cloud One (class DedicatedCloud) Monthly Cost: $50.00
Base Storage Cost: $40.00
Server Cost: $10.00

Cloud Two (class SharedCloud) Monthly Cost: $9.00
Base Storage Cost: $9.00
Data Stored: 12.000 GB
Data Limit: 20.000 GB
Overage: 0.000 GB
Cost Factor: 1.0

Cloud Three (class SharedCloud) Monthly Cost: $14.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 1.0

Cloud Four (class PublicCloud) Monthly Cost: $19.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
```

```
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 2.0

Cloud Five (class PersonalCloud) Monthly Cost: $12.00
Base Storage Cost: $9.00
Data Stored: 21.000 GB
Data Limit: 20.000 GB
Overage: 1.000 GB
Cost Factor: 3.0

----------------------------------------
Monthly Cloud Storage Report (by Name)
----------------------------------------
Cloud Five (class PersonalCloud) Monthly Cost: $12.00
Base Storage Cost: $9.00
Data Stored: 21.000 GB
Data Limit: 20.000 GB
Overage: 1.000 GB
Cost Factor: 3.0

Cloud Four (class PublicCloud) Monthly Cost: $19.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 2.0

Cloud One (class DedicatedCloud) Monthly Cost: $50.00
Base Storage Cost: $40.00
Server Cost: $10.00

Cloud Three (class SharedCloud) Monthly Cost: $14.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 1.0

Cloud Two (class SharedCloud) Monthly Cost: $9.00
Base Storage Cost: $9.00
Data Stored: 12.000 GB
Data Limit: 20.000 GB
Overage: 0.000 GB
Cost Factor: 1.0

------------------------------------------------
Monthly Cloud Storage Report (by Monthly Cost)
------------------------------------------------
Cloud One (class DedicatedCloud) Monthly Cost: $50.00
Base Storage Cost: $40.00
Server Cost: $10.00

Cloud Four (class PublicCloud) Monthly Cost: $19.00
```

```
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 2.0

Cloud Three (class SharedCloud) Monthly Cost: $14.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 1.0

Cloud Five (class PersonalCloud) Monthly Cost: $12.00
Base Storage Cost: $9.00
Data Stored: 21.000 GB
Data Limit: 20.000 GB
Overage: 1.000 GB
Cost Factor: 3.0

Cloud Two (class SharedCloud) Monthly Cost: $9.00
Base Storage Cost: $9.00
Data Stored: 12.000 GB
Data Limit: 20.000 GB
Overage: 0.000 GB
Cost Factor: 1.0

----------------------
Invalid Records Report
----------------------
S,Cloud Two B,9.0,a12.0,20.0
java.lang.NumberFormatException: For input string: "a12.0"

S,Cloud Three B,9.0,25.0
java.util.NoSuchElementException: For missing input data

Z,Cloud Zero,20.0,15.0,10.0
InvalidCategoryException: For category: Z


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

Notes

1. This project assumes that you are reading each double value as a String using next() and then parsing it into a double with Double.parseDouble(...) as shown in the following example.

   . . . Double.parseDouble(myInput.next());

   This form of input will throw a java.lang.NumberFormatException if the value is not a double.

   If you are reading in each double value as a double using nextDouble(), for example

   . . . myInput.nextDouble();

   then a java.util.InputMismatchException will be thrown if the value read in is not a double.

   For this assignment, you should change your input to use Double.parseDouble(...) rather than nextDouble(), since Web-CAT is looking for NumberFormatException rather than java.util.InputMismatchException.

2. If you are using the JUnit Assert.assertArrayEquals method to check two CloudStorage arrays for equality, then the `equals` and `hashCode` methods must be implemented in your CloudStorage class; that is, Assert.assertArrayEquals calls equals(Object obj) on each object in the array, so CloudStorage must have an equals method that overrides the one inherited from the Object class. If the CloudStorage class does not override equals(Object obj), then the JUnit Assert.assertArrayEquals method will use the inherited equals(Object obj) method which means two CloudStorage arrays will be equal only if they are the same object (i.e., aliases).

   Below is a simplified `equals` method and `hashCode` method you are free to use.

```
public boolean equals(Object obj) {
   if (!(obj instanceof CloudStorage)) {
      return false;
   }
   else {
      CloudStorage c = (CloudStorage) obj;
      return (name.equalsIgnoreCase(c.getName())); `
   }
}

public int hashCode() {
   return 0;
}
```