Update 11/25/2013 - See changes regarding main method (and testing main) highlighted below (pages 1, 4, 5, 6, 7)

Project 11: CellPhones – Part 3

Due: Completed Code – **Thursday, December 5, 2013 by 11:59 p.m.** (Web-CAT)

Deliverables

Your project files should be submitted to Web-CAT by the due date and time specified above (see the Lab Guidelines for information on submitting project files). You must submit your completed code files to Web-CAT no later than 11:59 PM on the due date for the completed code to avoid a late penalty for the project. You may submit your completed code up to 24 hours after the due date, but there is a late penalty of 15 points. No projects will be accepted after the one day late period. If you are unable to submit via Web-CAT, you should e-mail your project Java files in a zip file to your lab instructor before the deadline.

Files to submit to Web-CAT (submission of test files is optional):

From Project 9

- CellPhone.java, CellPhoneTest.java [modify constructor to throw NegativeValueException]
- FlipPhone.java, FlipPhoneTest.java
- SmartPhone.java, SmartPhoneTest.java [modify constructor to throw NegativeValueException]
- IPhone.java, IPhoneTest.java [modify constructor to throw NegativeValueException]
- Android.java,AndroidTest.java [modify constructor to throw NegativeValueException]

From Project 10

- Provider.java, ProviderTest.java [must be modified as described below]
- CellPhoneBillComparator.java, CellPhoneBillComparatorTest.java

New in Project 11

- NegativeValueException.java, NegativeValueExceptionTest.java
- CellPhonesPart3.java, ~~no test file required for this class~~ CellPhonesPart3Test.java

Recommendations

You should create new folder for Project 11 and copy your relevant Project 9 and 10 source files to it (i.e., do not include CellPhonesPart1.java and CellPhonesPart2.java). You should create a jGRASP project and add these source files as well as those created in Project 11. You may find it helpful to use the "viewer canvas" feature in jGRASP as you develop and debug your program.

Specifications

Overview: This project is Part 3 of three that involves calculating the bills for cell phones. In Part 1, you developed Java classes that represent categories of cell phones including flip phones, generic smart phones, iPhones, and Android phones. In Part 2, you implemented three additional classes: (1) CellPhoneBillComparator that implements the Comparator interface, (2) Provider that represents a provider of cell phones services and includes several specialized methods, and (3) CellPhonesPart2 which contains the main method for the program. In Part 3 (Project 11), you are to add exception handing and invalid input reporting. You will need to do the following: (1) create a new class named NegativeValueException which extends the Exception class, (2) add try-catch statements to catch IOException in the main method of the CellPhonesPart3 class, and (3) modify the readCellPhoneFile in the Provider class to catch/handle NegativeValueException, NumberFormatException, and NoSuchElementException in the event that these type exceptions are thrown while reading the input file.

Note that the main method in CellPhonesPart3 should create a Provider object and then invoke the readCellPhoneFile method on the Provider object to read data from a file and add cell phones to the team. You can use CellPhonesPart3 in conjunction with interactions by running the program in the 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. You should create a jGRASP project upfront and then add the source files as they are created. All of your files should be in a single folder.

- **CellPhone, FlipPhone, SmartPhone, IPhone, Android, CellPhoneBillComparator**

  Requirements and Design: No changes from the specifications in Projects 9 and 10.

- **NegativeValueException.java**

  Requirements and Design: NegativeValueException is a user defined exception created by extending the Exception class. This exception is to be caught in the readCellPhoneFile method in the Provider class when a line of input data contains a negative value for one of the numeric input values. The NegativeValueException is to be thrown in the cell phone constructor that is responsible for setting the field in question. The constructor for NegativeValueException takes a single String parameter representing the *message* and invokes the super constructor with the String. The following shows how the constructor would be called.

```
new NegativeValueException("All values must be positive")
```

  This string parameter will be the toString() value of a NegativeValueException when it occurs. For a similar constructor, see InvalidLengthException.java in 11_Exceptions\Examples\Polygons from this week's lecture notes.

  Testing: Here is an example of a test method that checks to make sure a negative value for *texts* in the constructor for FlipPhone throws a NegativeValueException. You should consider adding test methods to check for negative values for the other numeric fields.

```
@Test public void NegativeValueExceptionTest2() {
   boolean thrown = false;
   try {
      FlipPhone fp = new FlipPhone("123-456-7890", -100, 200);
   }
   catch (NegativeValueException e) {
      thrown = true;
   }
   Assert.assertTrue(thrown);
}
```

- **Provider.java**

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

  Design: In addition to the specifications in Project 10, the existing readCellPhoneFile method must be modified to catch following: NegativeValueException, NumberFormatException, and NoSuchElementException. When these exceptions occur, an appropriate message along with the offending line/record should be added the excludedRecords array.

  - `readCellPhoneFile` has no return value and accepts the data file name as a String, and throws IOException. This method creates a Scanner object to read in the file and then reads it in line by line. The first line contains the provider name and each of the remaining lines contains the data for a cell phone. After reading in the provider name, the "cell phone" lines should be processed as follows. A cell phone line is read in, a second scanner is created on the line, and the individual values for the cell phone are read in. After the values on the line have been read in, an "appropriate" cell phone object is created and added to the phones array using the addPhone method. If the cell phone type is not recognized, the record/line should be added to the excluded records array using the addExcludedRecord method. The data file is a "comma separated values" file; i.e., if a line contains multiple values, the values are delimited by commas. So when you set up the scanner for the cell phone lines, you need to set the delimiter to use a ",". Each cell phone line in the file begins with a category for the cell phone. Your switch statement should determine which type of CellPhone to create based on the first character of the category (i.e., **F**, **S**, **I**, and **A** for **F**lipPhone, **S**martPhone, **I**Phone, and **A**ndroid respectively). The second field in the record is the phone number, followed by the values for texts, minutes, data, iMessages, and hotspot minutes as appropriate for the category of phone. That is, the items that follow minutes correspond to the data needed for the particular category (or subclass) of CellPhone. For each *incorrect* line scanned (i.e., a line of data that contain missing data or invalid numeric data), your method will need to handle the invalid items properly. If a line includes invalid numeric data (e.g., the value for *texts*, an int, contains an alphabetic character or a decimal point), a NumberFormatException (see notes on last page) will be thrown automatically by the Java Runtime Environment (JRE). Your readCellPhoneFile method should catch and handle NumberFormatException, NoSuchElementException (for missing values), and NegativeValueException (thrown in cell phone constructors when a negative value is passed in) as follows. In each catch clause, a String should be created consisting of the exception, its initials in parenthesis, and the line with the invalid data. For example, if e is a NumberFormatException, the String resulting from the following expression should be added to the excludedRecords array.

    ```
    e + " (NFE) in: " + line
    ```

    For a NoSuchElementException e, the following String should be added to the excludedRecords array.

    ```
    e + " (NSEE) in: " + line
    ```

    For a NegativeValueException e, the following String should be added to the excludedRecords array.

    ```
    e + " (NVE) in: " + line
    ```

The file *provider3.csv* is available for download from the course web site. Below are example data records (the first line/record containing the provider name is followed by cell phone lines/records):

```
AU Cellular Service
FlipPhone,111-243-5948,100,50
BrickPhone,111-534-5948,100,50
FlipPhone,111-243-6924,30,77.0
Android,111-934-9939,500,400,1000,30
Iphone,111-123-4567,20,548,220,55
SmartPhone(Windows),111-131-3131,40,21,10
FlipPhone,111-342-7544,34,955
FlipPhone,111-243-6924,thirty,77
SmartPhone(Windows),111-869-5559,34,114,.**^&
Android,111-443-4434,50,430,2479,xyz
Iphone,111-111-1111,1,1,1,one
FlipPhone
FlipPhone,111-244-2192
FlipPhone,111-244-2193,29
SmartPhone
SmartPhone(Windows),111-135-2468
SmartPhone(Windows),111-135-2467,39
SmartPhone(Windows),111-135-2466,48,34
Iphone
Iphone,111-293-2303
Iphone,111-293-2304,6
Iphone,111-293-2305,6,7
Iphone,111-293-2306,7,8,9
Android
Android,111-955-1111
Android,111-955-1111,60
Android,111-955-1111,77,30
Android,111-955-1111,30,40,50
Android,111-566-4533,-1,23,50,100
FlipPhone,111-243-6924,30,-40
FlipPhone,111-243-6925,-30,40
SmartPhone(Windows),111-131-3333,-40,11,35
Android,111-324-4342,101,678,145,-30
Iphone,111-123-5435,5,5,5,-55
```

- **CellPhonesPart3.java**

Requirements and Design: The CellPhonesPart3 class has only a main method as described below. In addition to the specifications in Project 10, the main method should be modified to catch and handle an IOException if one is thrown in the readCellPhoneFile method.

In Part 3, `main` reads in the file name ~~from the user rather than~~ from the command line as was done in Project 10, creates an instance of Provider, and then calls the readCellPhoneFile method in the Provider class to read in the data file. After successfully reading in the file, the main method then prints the summary, rates, cell phone list by number, cell phone list by billing amount, and the list of excluded records. The main method should not include the *throws IOException* in the declaration. Instead, the main method should include a try-catch statement to catch IOException when/if it is thrown in the readCellPhoneFile method in the Provider class. This exception will occur when an incorrect file name is passed to the readCellPhoneFile method. After this

exception is caught, print the messages below and end. ~~Then your program should loop back and ask the user to re-enter the file name.~~ ~~**Your program should continue to attempt to read in a file name until either (1) the user presses ENTER without keying in any text (i.e., a String of length 0) or (2) the readCellPhoneFile method completes normally**~~. ~~One possible approach for the body of the loop is a follows. After the file name is read in from System.in, you can use an *if* statement to check to see if the file name is a String of length 0. If true, a return statement can be used to exit the loop and main and thus terminate the program without attempting to read the file and without printing any reports. Otherwise, the file is read within a *try-catch* statement. Below is an example where an incorrect file name. On the second try, ENTER was pressed without keying in any file name.~~

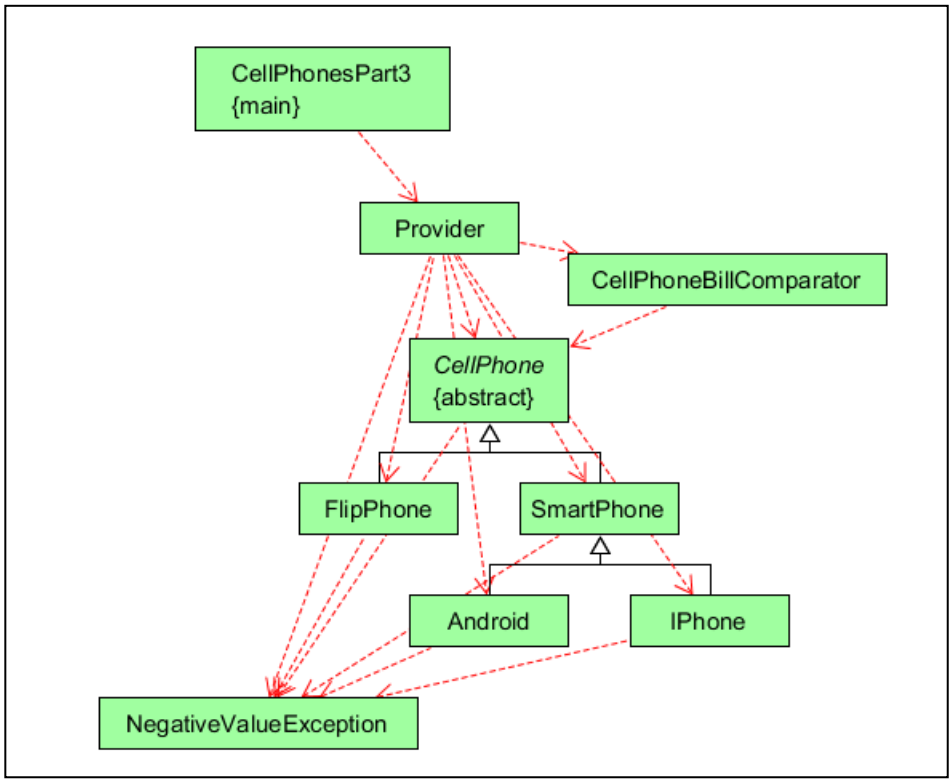
```
~~Enter file name: hello.csv~~
*** File not found.
~~Re-enter file name:~~
Program ending.
```

Also, if the user runs the program without a command line argument (e.g., `args.length == 0`), main should print the following message and end.

```
*** File name not provided by command line argument.
Program ending.
```

An example data file can be downloaded from the Lab web page. The program output for *provider3.csv* begins on the next page. See note below for testing your main method.

UML Class Diagram



Example Output for provider3.csv

```
----jGRASP exec: java -ea CellPhonesPart3

Enter file name: provider3.csv

------------------------------
Summary for AU Cellular Service
------------------------------
Number of cell phones: 5
Texts: 694
Talk Minutes: 1974
Data 1230
Hotspot Minutes: 30
iMessages: 55
Bill Total: $664.40

------------------------------
Rates for AU Cellular Service
------------------------------
FlipPhone Talk Rate: $0.15   Text Rate: $0.25
SmartPhone Talk Rate: $0.10   Text Rate: $0.50   Max Talk Time: 600.0
   iPhone iMessage Rate: $0.35
   Android Hotspot Rate: $0.75

------------------------------
Cell Phones By Number
------------------------------
Number: 111-123-4567 (IPhone)
Bill: $95.05 for 20 Texts, 548 Talk Minutes, 220 MB of Data, 55 iMessages

Number: 111-131-3131 (SmartPhone)
Bill: $22.60 for 40 Texts, 21 Talk Minutes, 10 MB of Data

Number: 111-243-5948 (FlipPhone)
Bill: $32.50 for 100 Texts, 50 Talk Minutes

Number: 111-342-7544 (FlipPhone)
Bill: $151.75 for 34 Texts, 955 Talk Minutes

Number: 111-934-9939 (Android)
Bill: $362.50 for 500 Texts, 400 Talk Minutes, 1000 MB of Data, 30 Hotspot Minutes



------------------------------
Cell Phones By Billing Amount
------------------------------

Number: 111-131-3131 (SmartPhone)
Bill: $22.60 for 40 Texts, 21 Talk Minutes, 10 MB of Data

Number: 111-243-5948 (FlipPhone)
Bill: $32.50 for 100 Texts, 50 Talk Minutes

Number: 111-123-4567 (IPhone)
Bill: $95.05 for 20 Texts, 548 Talk Minutes, 220 MB of Data, 55 iMessages

Number: 111-342-7544 (FlipPhone)
Bill: $151.75 for 34 Texts, 955 Talk Minutes

Number: 111-934-9939 (Android)
Bill: $362.50 for 500 Texts, 400 Talk Minutes, 1000 MB of Data, 30 Hotspot Minutes



------------------------------
Excluded Records
------------------------------
Invalid Category: BrickPhone,111-534-5948,100,50
java.lang.NumberFormatException: For input string: "77.0" (NFE) in: FlipPhone,111-243-6924,30,77.0
java.lang.NumberFormatException: For input string: "thirty" (NFE) in: FlipPhone,111-243-6924,thirty,77
java.lang.NumberFormatException: For input string: ".**^&" (NFE) in: SmartPhone(Windows),111-869-5559,34,114,.**^&
java.lang.NumberFormatException: For input string: "xyz" (NFE) in: Android,111-443-4434,50,430,2479,xyz
java.lang.NumberFormatException: For input string: "one" (NFE) in: Iphone,111-111-1111,1,1,1,one
java.util.NoSuchElementException (NSEE) in: FlipPhone
java.util.NoSuchElementException (NSEE) in: FlipPhone,111-244-2192
java.util.NoSuchElementException (NSEE) in: FlipPhone,111-244-2193,29
```

```
java.util.NoSuchElementException (NSEE) in: SmartPhone
java.util.NoSuchElementException (NSEE) in: SmartPhone(Windows),111-135-2468
java.util.NoSuchElementException (NSEE) in: SmartPhone(Windows),111-135-2467,39
java.util.NoSuchElementException (NSEE) in: SmartPhone(Windows),111-135-2466,48,34
java.util.NoSuchElementException (NSEE) in: Iphone
java.util.NoSuchElementException (NSEE) in: Iphone,111-293-2303
java.util.NoSuchElementException (NSEE) in: Iphone,111-293-2304,6
java.util.NoSuchElementException (NSEE) in: Iphone,111-293-2305,6,7
java.util.NoSuchElementException (NSEE) in: Iphone,111-293-2306,7,8,9
java.util.NoSuchElementException (NSEE) in: Android
java.util.NoSuchElementException (NSEE) in: Android,111-955-1111
java.util.NoSuchElementException (NSEE) in: Android,111-955-1111,60
java.util.NoSuchElementException (NSEE) in: Android,111-955-1111,77,30
java.util.NoSuchElementException (NSEE) in: Android,111-955-1111,30,40,50
NegativeValueException: All values must be positive (NVE) in: Android,111-566-4533,-1,23,50,100
NegativeValueException: All values must be positive (NVE) in: FlipPhone,111-243-6924,30,-40
NegativeValueException: All values must be positive (NVE) in: FlipPhone,111-243-6925,-30,40
NegativeValueException: All values must be positive (NVE) in: SmartPhone(Windows),111-131-3333,-40,11,35
NegativeValueException: All values must be positive (NVE) in: Android,111-324-4342,101,678,145,-30
NegativeValueException: All values must be positive (NVE) in: Iphone,111-123-5435,5,5,5,-55


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

Notes:

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

   ```
   . . . Integer.parseInt(myInput.next());
   ```

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

   If you are reading in each int value as a int using nextInt(), for example

   ```
   . . . myInput.nextInt();
   ```

   then a java.util.InputMismatchException will be thrown if the value read is not an int. Since an InputMismatchException is a subclass of NoSuchElementException, this exception will be caught in your catch clause for NoSuchElementException but will be reported as an InputMismatchException.

   You can either change your input to use Integer.parseInt(...) or you can catch the java.util.InputMismatchException and handle it the same way you handled the NumberFormatException but use the initials IME as follows.

   ```
   e + " (IME) in: " + line
   ```

   If you have mixed the two forms of input in your program and you want to keep both, then you will need to catch and handle both of the exceptions.

2. **Testing your main method**: You will need three test methods for CellPhonesPart3Test.java: (1) test with a good file name, (2) test with a bad file name, and (3) test with no file name (i.e., the user did not a command line argument).