

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. The grades for the Completed Code submission will be determined by the tests that you pass or fail in your test files and by the level of coverage attained in your source files as well as usual correctness tests in Web-CAT.

Files to submit to Web-CAT:

From Wireless Networks – Part 1

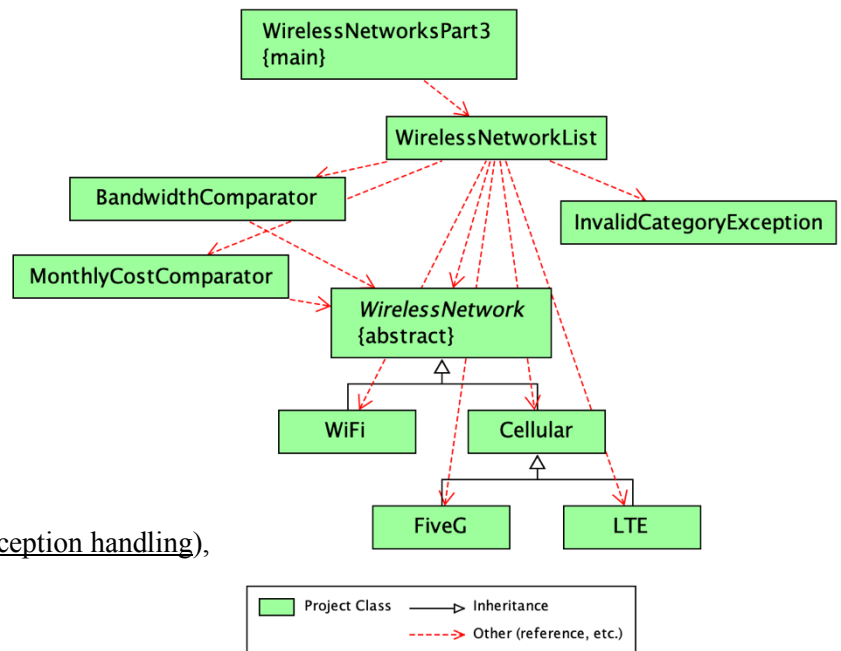
- WirelessNetwork.java
- WiFi.java, WiFiTest.java
- Cellular.java, CellularTest.java
- LTE.java, LTETest.java
- FiveG.java, FiveGTest.java

From Wireless Networks – Part 2

- BandwidthComparator.java, BandwidthComparatorTest.java
- MonthlyCostComparator.java, MonthlyCostComparatorTest.java
- WirelessNetworkList.java (add exception handling), WirelessNetworkListTest.java

New in Wireless Networks – Part 3

- WirelessNetworksPart3.java (with exception handling), WirelessNetworksPart3Test.java
- InvalidCategoryException



Recommendations

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

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

Overview: Wireless Networks – Part 3 is the third of a three-part software project that involves the monthly cost and reporting for wireless networks. 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. The main method in the **WirelessNetworksPart3**

class, which reads in the file name from the command line, 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 `WirelessNetworkList` will need to handle exceptions that occur while processing the data file, including a new exception called `InvalidCategoryException`.

WiFi, Cellular, LTE, and FiveG, `BandwidthComparator`, and `MonthlyCostComparator`

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

`WirelessNetworkList.java`

Requirements: The `WirelessNetworkList` class provides methods for reading in the data file and generating the reports. The `readFile` method should be redesigned to handle exceptions in the data. Reading a “good” line of data results in a new element added to the `WirelessNetwork` array, and reading a “bad” line of data results in the line + an exception message being added 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 `WirelessNetworkList` class has fields, a constructor, and methods as outlined below.

- **Fields:** no change from Part 2.
- **Constructor:** no change from Part 2.
- **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 `WirelessNetwork` object of the appropriate category (subclass) is added to the `WirelessNetwork` 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 W, C, L, or F. 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 `WirelessNetworksPart3` 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 `wireless_network_data2_exceptions.csv` from the assignment page in Canvas to test your program. Your JUnit test methods should force the exceptions described above to be 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.

WirelessNetworksPart3.java

Requirements: The `WirelessNetworksPart3` 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 `WirelessNetworksPart3` 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 `WirelessNetworkList` object, and then invokes its methods to (1) read the file and process the wireless network records and (2) to generate and print the five 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 `WirelessNetworkList` 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 exist, 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 `wireless_network_data2_exceptions.csv` from the assignment page in Canvas to test your program. One of your 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 this 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 `WirelessNetworksPart2` 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 WirelessNetworksPart3
File name expected as command line argument.
Program ending.

----jGRASP: operation complete.

----jGRASP exec: java WirelessNetworksPart3 nofile.csv
*** Attempted to read file: nofile.csv (No such file or directory)

----jGRASP: operation complete.

----jGRASP exec: java WirelessNetworksPart3 wireless_network_data2.exceptions.csv
-----
Monthly Wireless Network Report
-----
My Wifi (class WiFi) Cost: $45.00
Bandwidth: 450.0 Mbps

My Note (class Cellular) Cost: $20.00
Bandwidth: 5.0 Mbps
Time: 1200.0 seconds
Data Limit: 1.0 GB
Data Used: 0.75 GB

My iPad (class LTE) Cost: $38.00
Bandwidth: 20.0 Mbps
Time: 1200.0 seconds
Data Limit: 2.0 GB
Data Used: 3.0 GB

My Phone (class FiveG) Cost: $80.00
Bandwidth: 80.0 Mbps
Time: 1200.0 seconds
Data Limit: 10.0 GB
Data Used: 12.0 GB
```

Monthly Wireless Network Report (by Name)

My iPad (class LTE) Cost: \$38.00
Bandwidth: 20.0 Mbps
Time: 1200.0 seconds
Data Limit: 2.0 GB
Data Used: 3.0 GB

My Note (class Cellular) Cost: \$20.00
Bandwidth: 5.0 Mbps
Time: 1200.0 seconds
Data Limit: 1.0 GB
Data Used: 0.75 GB

My Phone (class FiveG) Cost: \$80.00
Bandwidth: 80.0 Mbps
Time: 1200.0 seconds
Data Limit: 10.0 GB
Data Used: 12.0 GB

My Wifi (class WiFi) Cost: \$45.00
Bandwidth: 450.0 Mbps

Monthly Wireless Network Report (by Bandwidth)

My Note (class Cellular) Cost: \$20.00
Bandwidth: 5.0 Mbps
Time: 1200.0 seconds
Data Limit: 1.0 GB
Data Used: 0.75 GB

My iPad (class LTE) Cost: \$38.00
Bandwidth: 20.0 Mbps
Time: 1200.0 seconds
Data Limit: 2.0 GB
Data Used: 3.0 GB

My Phone (class FiveG) Cost: \$80.00
Bandwidth: 80.0 Mbps
Time: 1200.0 seconds
Data Limit: 10.0 GB
Data Used: 12.0 GB

My Wifi (class WiFi) Cost: \$45.00
Bandwidth: 450.0 Mbps

Monthly Wireless Network Report (by Monthly Cost)

My Phone (class FiveG) Cost: \$80.00
Bandwidth: 80.0 Mbps
Time: 1200.0 seconds

```
Data Limit: 10.0 GB
Data Used: 12.0 GB

My Wifi (class WiFi) Cost: $45.00
Bandwidth: 450.0 Mbps

My iPad (class LTE) Cost: $38.00
Bandwidth: 20.0 Mbps
Time: 1200.0 seconds
Data Limit: 2.0 GB
Data Used: 3.0 GB

My Note (class Cellular) Cost: $20.00
Bandwidth: 5.0 Mbps
Time: 1200.0 seconds
Data Limit: 1.0 GB
Data Used: 0.75 GB

-----
Invalid Records Report
-----
X,Bad Data,0,0,0,0,0,0,0
InvalidCategoryException: For category: X

W,My Other Wifi,450,40.0a,5.00
java.lang.NumberFormatException: For input string: "40.0a"

C,My Other Note,5.0,20.00,1200
java.util.NoSuchElementException: For missing input data

----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.assertEquals` method to check two `WirelessNetwork` arrays for equality, then the `equals` and `hashCode` methods must be implemented in your `WirelessNetwork` class; that is, `Assert.assertEquals` calls `equals(Object obj)` on each object in the array, so `WirelessNetwork` must have an `equals` method that overrides the one inherited from the `Object` class. If the `WirelessNetwork` class does not override `equals(Object obj)`, then the JUnit `Assert.assertEquals` method will use the inherited `equals(Object obj)` method which means two `WirelessNetwork` 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 WirelessNetwork)) {
        return false;
    }
    else {
        WirelessNetwork c = (WirelessNetwork) obj;
        return (name.equalsIgnoreCase(c.getName()));
    }
}

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