# Project: Wireless Networks – Part 2



## 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.

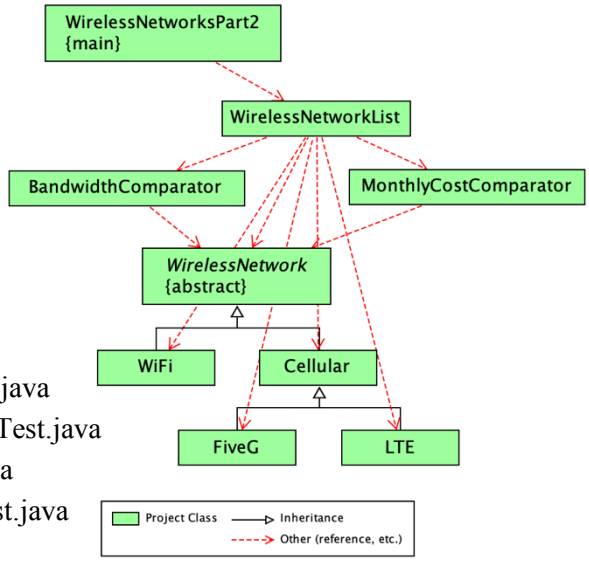
Files to submit to Web-CAT:

From Wireless Networks – Part 1

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

New in Wireless Networks – Part 2

- BandwidthComparator.java, BandwidthComparatorTest.java
- MonthlyCostComparator.java, MonthlyCostComparatorTest.java
- WirelessNetworkList.java, WirelessNetworkListTest.java
- WirelessNetworksPart2.java, WirelessNetworksPart2Test.java




## Recommendations

You should create new folder for Part 2 and copy your relevant Part 1 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**: This project is the second of three that will involve the monthly cost and reporting for wireless networks. In Part 1 you developed Java classes that represent categories of wireless networks including WiFi, cellular, LTE, and 5G. In Part 2, you will implement four additional classes: (1) BandwidthComparator that implements the Comparator interface for WirelessNetwork, (2) MonthlyCostComparator that implements the Comparator interface for WirelessNetwork, (3) WirelessNetworkList that represents a list of wireless networks and includes several specialized methods, and (4) WirelessNetworksPart2 which contains the main method for the program. Note that the main method in WirelessNetworksPart2 should create a WirelessNetworkList object and then call the readFile method on the WirelessNetworkList object, which will add wireless networks to the list as the data is read in from a file. You can use WirelessNetworksPart2 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 will create a 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 test files as they are created. All of your files should be in a single folder.

- **WiFi, Cellular, LTE, and FiveG**

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

- **WirelessNetwork.java**

  **Requirements and Design**: In addition to the specifications in Part 1, the WirelessNetwork class should implement the Comparable interface for WirelessNetwork, which means the following method must be implemented in WirelessNetwork.
  - `compareTo`: Takes a WirelessNetwork as a parameter and returns an int indicating the results of comparing the two WirelessNetwork object based on their respective name fields.

- **WirelessNetworkList.java**

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

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

  (1) **Fields:** 1) An array of WirelessNetwork 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 elements in the array. These two fields should be private.
  (2) **Constructor:** The constructor has no parameters and initializes the WirelessNetwork 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 WirelessNetworkList are described below.
    - `getWirelessNetworksArray` returns an array of type WirelessNetwork representing the WirelessNetwork array field.
    - `getInvalidRecordsArray` returns an array of type String representing the invalid records array field.
    - `addWirelessNetwork` has no return value, accepts a WirelessNetwork object, increases the capacity of the WirelessNetwork array by one, and adds the WirelessNetwork object in the last position of the WirelessNetwork 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 Project 11, 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 WirelessNetwork object is created and added to the WirelessNetwork 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, bandwidth, fixed monthly cost, followed by one or more fields specific to the category. Remember, WiFi, Cellular, LTE, and FiveG objects are all WirelessNetwork objects. The category codes are W for WiFi, C for Cellular, L for LTE, and F for FiveG. Any other category code is invalid. Below are examples data records:

```
W,My Wifi,450,40.00,5.00
C,My Note,5.0,20.00,1200,1.0
X,Bad Data,0,0,0,0,0,0,0
L,My iPad,20.0,30.00,1200,2.0
F,My Phone,80.0,50.00,1200,10.0
```

- `generateReport` processes the WirelessNetwork array using the original order from the file to produce the Monthly Wireless Network Report and then returns the report as String. See example result in output for WirelessNetworksPart2 beginning on page 6.
- `generateReportByName` sorts the WirelessNetwork array by its natural ordering, and processes the WirelessNetwork array to produce the Monthly Wireless Network Report (by Name), then returns the report as a String. See example result in output for WirelessNetworksPart2 beginning on page 6.
- `generateReportByBandwidth` sorts the WirelessNetwork array by its bandwidth, and processes the WirelessNetwork array to produce the Monthly Wireless Network Report (by Bandwidth), then returns the report as a String. See example result in output for WirelessNetworksPart2 beginning on page 6.
- `generateReportByMonthlyCost` sorts the WirelessNetwork array by monthly cost, and processes the WirelessNetwork array to produce the Monthly Wireless Network Report (by Monthly Cost) and then returns the report as String. See example result in output for WirelessNetworksPart2 beginning on page 6.

**Code and Test:** See examples of file reading and sorting (using Arrays.sort) in the class notes.

The natural sorting order based on a wireless network's name for is determined by the compareTo method when the Comparable interface is implemented.

```
Arrays.sort(getWirelessNetworksArray());
```

The sorting order based on a wireless network's bandwidth is determined by the BandwidthComparator class which implements the Comparator interface (described below).

```
Arrays.sort(getWirelessNetworksArray(), new BandwidthComparator());
```

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

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

In your JUnit 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));
```

- **BandwidthComparator.java**

  **Requirements and Design:** The BandwidthComparator class implements the Comparator interface for WirelessNetwork objects. Hence, it implements the method `compare(WirelessNetwork n1, WirelessNetwork n2)` that defines the ordering from lowest to highest based on the wireless network's bandwidth. See examples in class notes.

- **MonthlyCostComparator.java**

  **Requirements and Design:** The MonthlyCostComparator class implements the Comparator interface for WirelessNetwork objects. Hence, it implements the method `compare(WirelessNetwork n1, WirelessNetwork n2)` that defines the ordering from highest to lowest based on the wireless network's monthly cost. See examples in class notes.

- **WirelessNetworksPart2.java**

  **Requirements:** The WirelessNetworksPart2 class contains the main method for running the program.

  **Design:** The WirelessNetworksPart2 class is the driver class and has a main method described below.

  - `main` accepts a file name as a command line argument, creates a WirelessNetworkList object, and then invokes its methods to read the file and process the wireless network records and then to generate and print the four reports as shown in the example output beginning on page 6. 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:** In your JUnit test file for the WirelessNetworksPart2 class, you should have at least two test methods for the main method. One test method should invoke WirelessNetworksPart2.main(args) where args is an empty String array, and the other test method should invoke WirelessNetworksPart2.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 Cellular, you could assert that `Cellular.COST_FACTOR` equals `1.0` in each test methods.

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 WirelessNetworksPart2, the following should exercise
// the code for true.
String[] args1 = {};  // an empty String[]
WirelessNetworksPart2.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 = {"wireless_network_data1.csv"};
// args2[0] is the file name
WirelessNetworksPart2.main(args2);
```

If Web-CAT complains the default constructor for `WirelessNetworksPart2` has not been covered, you should include the following line of code in one of your test methods.

```
// to exercise the default constructor
WirelessNetworksPart2 app = new WirelessNetworksPart2();
```

**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 WirelessNetworksPart2
File name expected as command line argument.
Program ending.

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

## Example Output for *wireless_network_data1.csv*

```
 ----jGRASP exec: java WirelessNetworksPart2 wireless_network_data1.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


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

## Hints

1. Adding an element to a full array in your **addWirelessNetwork** 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 'W':` rather than `case "W":`). 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)
```