Project: Wireless Networks – Part 1

Deliverables

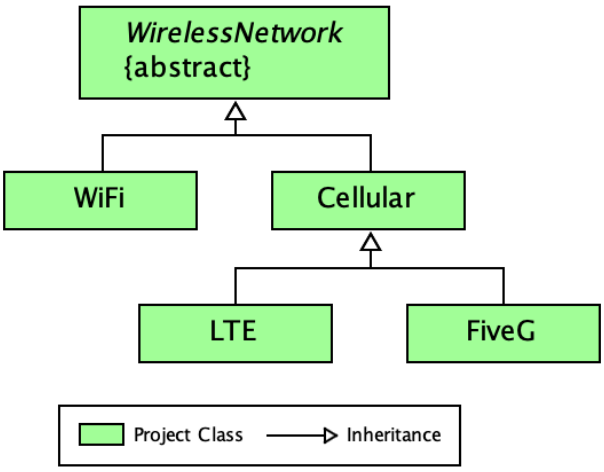
Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project). 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:

- WirelessNetwork.java
- WiFi.java, WiFiTest.java
- Cellular.java, CellularTest.java
- LTE.java, LTETest.java
- FiveG.java, FiveGTest.java
- (Optional) WirelessNetworksPart1.java, WirelessNetworksPart1Test.java

Specifications

Overview: This project is the first of three that will involve the monthly cost and reporting for wireless networks. You will develop Java classes that represent categories of wireless networks including WiFi, cellular, LTE, and 5G. You may also want to develop an optional driver class with a main method. As you develop each class, you should create the associated JUnit test file with the required test methods to ensure the classes and methods meet the specifications. You should create a jGRASP project upfront and then add the source and test files as they are created. All of your files should be in a single folder. Below is the UML class diagram for the required classes which shows the inheritance relationships.



You should read through the remainder of this assignment before you start coding.

- **WirelessNetwork.java**

  Requirements: Create an *abstract* WirelessNetwork class that stores wireless network data and provides methods to access the data.

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

  (1) **Fields**:
  instance variables (*protected*) for: (1) name of type String, (2) bandwidth of type double, and (3) monthly fixed cost of type double.
  class variable (*protected static*) for the count of WirelessNetwork objects that have been created; set to zero when declared and then incremented in the constructor.
  These are the only fields that this class should have.

  (2) **Constructor**: The WirelessNetwork class must contain a constructor that accepts three parameters representing the instance variables (name, bandwidth, and monthly fixed cost) and then assigns them as appropriate. Since this class is abstract, the constructor will be called from the subclasses of WirelessNetwork using *super* and the parameter list. The count field should be incremented in the constructor.

  (3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.
  - `getName`: Accepts no parameters and returns a String representing the name.
  - `setName`: Accepts a String representing the name, sets the field, and returns nothing.
  - `getBandwidth`: Accepts no parameters and returns a double representing the bandwidth in Mbps.
  - `setBandwidth`: Accepts a double representing the bandwidth in Mbps, sets the field, and returns nothing.
  - `getMonthlyFixedCost`: Accepts no parameters and returns a double representing monthly fixed cost.
  - `setMonthlyFixedCost`: Accepts a double representing the monthly fixed cost, sets the field, and returns nothing.
  - `getCount`: Accepts no parameters and returns an int representing the count. Since count is *static*, this method should be *static* as well.
  - `resetCount`: Accepts no parameters, resets count to zero, and returns nothing. Since count is *static*, this method should be *static* as well.
  - `toString`: Returns a String describing the WirelessNetwork object. This method will be inherited by the subclasses. For an example of the toString result, see the WiFi class and Cellular class below. Note that you can get the class name for an instance c by calling c.getClass() [or if inside the class, this.getClass()].

- `monthlyCost`: An *abstract* method that accepts no parameters and returns a double representing the monthly cost of a wireless network. Since this is abstract, each non-abstract subclass must implement this method.

Code and Test: Since the WirelessNetwork class is abstract you cannot create instances of WirelessNetwork upon which to call the methods. However, these methods will be inherited by the subclasses of WirelessNetwork. You should consider first writing skeleton code for the methods in order to compile WirelessNetwork so that you can create the first subclass described below. At this point you can begin completing the methods in WirelessNetwork and writing the JUnit test methods for your subclass that tests the methods in WirelessNetwork.

- **WiFi.java**

Requirements: Derive the class WiFi.java from WirelessNetwork.

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

(1) **Field**: *instance* variable for modemCost of type double. This variable should be declared with the *private* access modifier. This is the only field that should be declared in this class.

(2) **Constructor**: The WiFi class must contain a constructor that accepts four parameters representing the three instance fields in the WirelessNetwork class (name, bandwidth, and monthlyFixedCost) and the one instance field modemCost declared in WiFi. Since this class is a subclass of WirelessNetwork, the super constructor should be called with field values for WirelessNetwork. The instance variable modemCost should be set with the last parameter. Below is an example of how the constructor could be used to create an WiFi object:

```
WiFi n1 = new WiFi("My Wifi", 450, 40.00, 5.00);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `getModemCost`: Accepts no parameters and returns a double representing modemCost.
- `setModemCost`: Accepts a double representing the modemCost, sets the field, and returns nothing.
- `monthlyCost`: Accepts no parameters and returns a double representing the monthlyCost for the WiFi network calculated as the sum of monthly fixed cost and modem cost.
- **There is no `toString` method in this class**. When toString is invoked on an instance of WiFi, the toString method inherited from WirelessNetwork is called. Below is an example of the toString result for WiFi n1 as it is declared above.

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

Code and Test: As you implement the WiFi class, you should compile and test it as methods are created. Although you could use interactions, it should be more efficient to test by creating appropriate JUnit test methods. You can now continue developing the methods in

WirelessNetwork (parent class of WiFi). The test methods in WiFiTest should be used to test the methods in both WirelessNetwork and WiFi. Remember, WiFi *is-a* WirelessNetwork which means WiFi inherited the instance method defined in WirelessNetwork. Therefore, you can create instances of WiFi in order to test methods of the WirelessNetwork class. You may also consider developing WirelessNetworksPart1 (page 7) in parallel with this class to aid in testing.

- **Cellular.java**

Requirements: Derive the class Cellular from WirelessNetwork.

Design: The Cellular class has a field, a constructor, and methods as outlined below.

(1) **Fields:**
instance variables (*protected*) : (1) time of type double and (2) data limit of type double. These variables should be declared with the *protected* access modifier.
constant (*public static final*) COST_FACTOR of type double set to 1.0, which can be referenced as Cellular.COST_FACTOR.
These are the only field that should be declared in this class.

(2) **Constructor**: The Cellular class must contain a constructor that accepts five parameters representing the three instance fields in the WirelessNetwork class (name, bandwidth, and monthly fixed cost) and the two instance fields (time and data limit) declared in Cellular. Since this class is a subclass of WirelessNetwork, the super constructor should be called with field values for WirelessNetwork. The instance variables time and data limit should be set with the last two parameters. Below is an example of how the constructor could be used to create an Cellular object:

```
Cellular n2 = new Cellular("My Note", 5.0, 20.00, 1200, 1.0);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `getTime`: Accepts no parameters and returns a double representing time.
- `setTime`: Accepts a double representing the time in seconds, sets the field, and returns nothing.
- `getDataLimit`: Accepts no parameters and returns a double representing the data limit in GB.
- `setDataLimit`: Accepts a double representing the data limit in GB, sets the field, and returns nothing.
- `dataUsage`: Accepts no parameters and returns a double representing the data usage in GB for the network calculated as bandwidth / 8000 * time. Note that dividing by 8000 converts bandwidth in Mbps to GB.
- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the cellular network as follows. If data usage is less than or equal to the data limit, return the monthly fixed cost. Otherwise, return (monthly fixed cost + (data usage - data limit) * Cellular.COST_FACTOR).

- `toString`: Returns a String describing the Cellular object by calling parent's toString method, super.toString() and then appending the lines for time, data limit, and data usage. Below is an example of the toString result for Cellular n2 as it is declared above.

```
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
```

Code and Test: As you implement the Cellular class, you should compile and test it as methods are created. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Cellular in a JUnit test method in the CellularTest class and then run the test file. If you want to view your objects in the Canvas, set a breakpoint in your test method the run *Debug* on the test file. When it stops at the breakpoint, step until the object is created. Then open a canvas window using the canvas button at the top of the Debug tab. After you drag the instance onto the canvas, you can examine it for correctness. If you change the viewer to "toString" view, you can see the formatted toString value. You can also enter the object variable name in interactions and press ENTER to see the toString value. *Hint: If you use the same variable names for objects in the test methods, you can use the menu button on the viewer in the canvas to set "Scope Test" to "None". This will allow you to use the same canvas with multiple test methods*. You may also consider developing WirelessNetworksPart1 (page 7) in parallel with this class to aid in testing.

- **LTE.java**

Requirements: Derive the class LTE from Cellular.

Design: The LTE class has a field, a constructor, and methods as outlined below.

(1) **Field**: **constant (*public static final*)** COST_FACTOR of type double set to 4.0, which can be referenced as LTE.COST_FACTOR outside the class.
These are the only fields that should be declared in this class.

(2) **Constructor**: The LTE class must contain a constructor that accepts five parameters representing the three instance fields in the WirelessNetwork class (name, bandwidth, and monthly fixed cost) and the two instance fields (time and data limit) declared in Cellular. Below is an example of how the constructor could be used to create a LTE object:

```
LTE n3 = new LTE("My iPad", 20.0, 30.00, 1200, 2.0);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the LTE network as follows. If data usage is less than or equal the data limit, return the monthly fixed cost. Otherwise, return (monthly fixed cost + (data usage - data limit) * LTE.COST_FACTOR * 2).

- **There is no** `toString` **method in this class**. When toString is invoked on an instance of LTE, the toString method inherited from Cellular is called. Below is an example of the toString result for LTE n3 as it is declared above.

```
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
```

Code and Test: As you implement the LTE class, you should compile and test it as methods are created. For details, see **Code and Test** above for the WiFi and Cellular classes. You may also consider developing WirelessNetworksPart1 (page 7) in parallel with this class to aid in testing.

- **FiveG.java**

Requirements: Derive the class FiveG from class Cellular.

Design: The FiveG class has a field, a constructor, and methods as outlined below.

(1) **Field**: **constant (*public static final*)** COST_FACTOR of type double set to 5.0, which can be referenced as FiveG.COST_FACTOR outside the class.
This is the only field that should be declared in this class.

(2) **Constructor**: The FiveG class must contain a constructor that accepts five parameters representing the three instance fields in the WirelessNetwork class (name, bandwidth, and monthly fixed cost) and the two instance fields (time and data limit) declared in Cellular. Below is an example of how the constructor could be used to create a FiveG object:

```
FiveG n4 = new FiveG("My Phone", 80.0, 50.00, 1200, 10.0);
```

(3) **Methods**: Usually a class provides methods to access (or read) and modify each of its instance variables (known as get and set methods) along with any other required methods. At minimum you will need the following methods.

- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the FiveG network as follows. If data usage is less than or equal the data limit, return the monthly fixed cost. Otherwise, return (monthly fixed cost + (data usage - data limit) * FiveG.COST_FACTOR * 3).
- **There is no** `toString` **method in this class**. When toString is invoked on an instance of LTE, the toString method inherited from Cellular is called. Below is an example of the toString result for FiveG n4 as it is declared above.

```
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
```

Code and Test: As you implement the FiveG class, you should compile and test it as methods are created. For details, see **Code and Test** above for the WiFi and Cellular classes. You may also consider developing WirelessNetworksPart1 (page 7) in parallel with this class to aid in testing.

- **WirelessNetworksPart1.java (Optional)**
  Requirements: Driver class with main method is optional but you may find it helpful.

  Design: The WirelessNetworksPart1 class only has a main method as described below.

  The main method should be developed incrementally along with the classes above. For example, when you have compiled WirelessNetwork and WiFi, you can add statements to main that create and print an instance of WiFi. [Since WirelessNetwork is abstract you cannot create an instance of it.] When main is completed, it should contain statements that create and print instances of WiFi, Cellular, LTE, and FiveG. Since printing the objects will not show all of the details of the fields, you should also run WirelessNetworksPart1 in the canvas (or debugger with a breakpoint) to examine the objects. Between steps you can use interactions to invoke methods on the objects in the usual way. For example, if you create n1, n2, n3, and n4 as described in the sections above and your main method is stopped between steps after n4 has been created, you can enter the following in interactions to get the rating for the FiveG object.

  ```
  ► n4.monthlyCost()
  80.0
  ```

  The output from main assuming you create print the four objects n1, n2, n3, and n4 as described in the sections above is shown as below. Note that a new line was added in main before each object to achieve the spacing between objects.

  ```
  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
  ```

Code and Test: After you have implemented the WirelessNetworksPart1 class, you should create the test file WirelessNetworksPart1Test.java in the usual way. The only test method you need is one that checks the class variable *count* that was declared in WirelessNetwork and inherited by each subclass. In the test method, you should reset *count*, call your main method, then assert that *count* is four (assuming that your main creates four objects from the WirelessNetwork hierarchy). The following statements accomplish the test.

```
WirelessNetwork.resetCount();
WirelessNetworks1.main(null);
Assert.assertEquals("WirelessNetwork count should be 4. ",
                    4, WirelessNetwork.getCount());
```

Canvas for WirelessNetworkPart1

Below is an example of a jGRASP viewer canvas for WirelessNetworkPart1 that contains a viewer for the class variable WirelessNetwork.count and two viewers for each of n1, n2, n3, and n4. The first viewer for each is set to Basic viewer and the second is set to the toString viewer. The canvas was created dragging instances from the debug tab into a new canvas window and setting the appropriate viewer. Note that you will need to unfold one of the instances in the debug tab to find the static variable *count*.