COMP 1210 Project 9: Cell Phones – Part 1

Due: Skeleton Code (ungraded) **– Monday, November 11, 2013 by 11:59 p.m.**
Completed Code – **Monday, November 11, 2013 by 11:59 p.m.**

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 may submit your skeleton code files until Mon 11 Nov 2013 11:59 p.m. to check for compilation but not for credit. You should do your best to get this done in lab on Wed or by the end of the Help session on Fri to ensure that you are making appropriate progress. 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. The Completed Code will be tested against your test methods in your JUnit test files and against the usual correctness tests. The grade will be determined by the tests that you pass or fail and the level of coverage attained in your Java source files by your test methods.

Files to submit to Web-CAT:

- CellPhone.java, CellPhoneTest.java
- FlipPhone.java, FlipPhoneTest.java
- SmartPhone.java, SmartPhoneTest.java
- IPhone.java, IPhoneTest.java
- Android.java,AndroidTest.java
- CellPhonesPart1.java, CellPhonesPart1Test.java

Specifications

Overview: This project is the first of three that will involve calculating the bills for cell phones. The billing system is unique in that the rates are based on the type of cell phone used. You will develop Java classes that represent categories of cell phones including flip phones, generic smart phones, iPhones, and Android phones. All cell phones may make calls and send messages which are billed per minute and per message respectively. Smart phones add the capability to send and receive data. Calls are cheaper on smart phones, but become more expensive after a certain minute limit is exceeded. The iPhone has the ability to send and receive iMessages, which are sent via the Internet rather than SMS. These are more expensive than regular data, but cheaper than a regular text message. Android phones may create a mobile hotspot, allowing nearby devices to access mobile broadband. Note that CellPhonesPart1.java should contain a main method that creates one instance of each of the non-abstract classes in the CellPhone hierarchy. That is, as you develop each non-abstract class, you should add code in the main method to create and print an instance of the class. Thus, after you have created all the classes, your main method should create and print four objects (i.e., one each for FlipPhone, SmartPhone, IPhone, and Android). You can use CellPhonesPart1 in conjunction with interactions by running the program in the canvas (or debugger with a breakpoint) and single stepping until the each of the instances is created. You can then enter interactions for the instances 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. All of your files should be in a single folder. You should create a jGRASP project upfront and then add the source and test files as they are created. You should generate (or regenerate) the UML class diagram each time you add a class to the project.

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

- **CellPhone.java**

  Requirements: Create an *abstract* CellPhone class that stores cell phone data and provides methods to access the data.

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

  (1) **Fields**: Three *instance* variables for the phone's number of type String, texts of type int, and minutes of type int. These variables should be declared with the *protected* access modifier so that they are accessible in the subclasses of CellPhone. The fourth field should be a static variable, cellPhoneCount, of type int with private access. This class variable is used to track the number of cell phones that are created from the classes in the CellPhone hierarchy. These are the only fields that this class should have.

  (2) **Constructor**: The CellPhone class has a constructor that accepts three parameters representing the values for the respective *instance* variables, assigns the values, and then increments the cellPhoneCount class variable. Since this class is abstract, the constructor will be called from the subclasses of CellPhone using *super* and the parameter list.

  (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.
  - `getNumber`: Accepts no parameters and returns a String representing the phone's number.
  - `setNumber:` Accepts a String representing the number, sets the field, and returns nothing.
  - `getTexts`: Accepts no parameters and returns an int representing the texts.
  - `setTexts:` Accepts an int representing texts, sets the field, and returns nothing.
  - `getMinutes`: Accepts no parameters and returns an int representing minutes.
  - `setMinutes:` Accepts an int representing the minutes, sets the field, and returns nothing.
  - `getCellPhoneCount`: Accepts no parameters and returns an int representing the count. Since cellPhoneCount is *static*, this method should be *static* as well.
  - `resetCellPhoneCount:` Accepts no parameters, resets cellPhoneCount to zero, and returns nothing. Since count is *static*, this method should be *static* as well.

- `calculateBill`: An abstract method that accepts no parameters and returns a double representing the bill.
- `resetBill:` An abstract method that accepts no parameters and returns nothing.
- `toString`: Returns a String describing the CellPhone object. This method will be inherited by the subclasses although it may overridden in the subclass as appropriate. If it is overridden, then it may be called from the toString method in the subclasses of CellPhone using super.toString(). For an example of the toString result, see the FlipPhone class below. Note that you can get the class name for an instance c by calling `c.getClass().getName()` or `this.getClass().getName()` within the class.

- **FlipPhone.java**

Requirements: Derive the FlipPhone class from the CellPhone class.

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

(1) **Fields**: public constants for TALK_RATE = 0.15 and TEXT_RATE = 0.25 of type double. These are the only fields that should be declared in this class.

(2) **Constructor**: The FlipPhone class must contain a constructor that accepts three parameters representing the three instance fields in the CellPhone class. Since this class is a subclass of CellPhone, the super constructor should be called with field values for CellPhone. Below is an example of how the constructor could be used to create an FlipPhone object:

```
FlipPhone flip1 = new FlipPhone("123-456-7890", 100, 50);
```

(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.

- `calculateBill`: Accepts no parameters and returns a double representing bill calculated as follows: texts * TEXT_RATE + minutes * TALK_RATE
- `resetBill:` Accepts no parameters, sets the texts and minutes fields to zero, and returns nothing.
- `toString`: There is no toString method required in the FlipPhone class. Since no instance fields were added in FlipPhone, we can use the toString method inherited from CellPhone. That is, when toString is invoked on an instance of FlipPhone, the toString method inherited from CellPhone will be called. Below is an example of the toString result for FlipPhone flip1 as it is declared above.

```
Number: 123-456-7890 (FlipPhone)
Bill: $32.50 for 100 Texts, 50 Talk Minutes
```

Code and Test: As you implement the FlipPhone class, you should compile and test it as methods are created. Although you could use interactions, it may be more efficient to test: (1) by using the CellPhonesPart1 class and main method described below and/or (2) by creating appropriate JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of FlipPhone in the main method of

CellPhonesPart1 and then run CellPhonesPart1 in the canvas. After you drag the instance onto the canvas, you can examine it for correctness. As FlipPhone is implemented, you should begin creating test methods in the corresponding FlipPhoneTest.java file. Also you can now create or continue developing the test file for CellPhone (parent class of FlipPhone) and test the methods declared in it. Instances of CellPhone cannot be created, but a FlipPhone *is-a* CellPhone.

- **SmartPhone.java**

Requirements: Derive the SmartPhone class from the CellPhone class.

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

(1) **Field**: instance variable for data of type int, which should be declared with the *protected* access modifier; public constants of type double for TALK_RATE = 0.10, TEXT_RATE = 0.50, OVERTIME_TALK_RATE = 2.0, MAX_TALK_TIME = 600.0, and DATA_RATE = 0.05. These are the only fields that should be declared in this class.

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

```
SmartPhone smart1 = new SmartPhone("123-456-7891", 40, 21, 10);
```

(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.

- o `getData`: Accepts no parameters and returns an int representing the data field.
- o `setData:` Accepts an int representing the data, sets the field, and returns nothing.
- o `calculateBill`: Accepts no parameters and returns a double representing bill calculated as follows: TEXT_RATE * texts + DATA_RATE * data + the charge for minutes. The charge for minutes is calculated as minutes * TALK_RATE if minutes does not exceed MAX_TALK_TIME. If minutes exceeds MAX_TALK_TIME, charge for minutes is calculated as MAX_TALK_TIME * TALK_RATE + (minutes - MAX_TALK_TIME) * OVERTIME_TALK_RATE.
- o `resetBill:` Accepts no parameters, sets the texts, minutes, and data fields to zero, and returns nothing.
- o `toString`: The toString method in the SmartPhone class should invoke the toString method in the CellPhone and the concatenate the data information. Below is an example of the toString result for SmartPhone smart1 as it is declared above.

```
Number: 123-456-7891 (SmartPhone)
Bill: $22.60 for 40 Texts, 21 Talk Minutes, 10 MB of Data
```

Code and Test: As you implement the SmartPhone class, you should compile and test it as methods are created. Although you could use interactions, it may be more efficient to test: (1) by using the CellPhonesPart1 class and main method described below and/or (2) by creating appropriate JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of SmartPhone in the main method of CellPhonesPart1 and then run CellPhonesPart1 in the canvas. After you drag the instance onto the canvas, you can examine it for correctness. After you have implemented and compiled one or more of the methods, you should begin creating test methods in the corresponding SmartPhoneTest.java file.

- **IPhone.java**

Requirements: Derive the IPhone class from the SmartPhone class.

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

(1) **Field**: an *instance* variable for iMessages of type int, which should be declared with the *protected* access modifier; a public constant of type double for IMESSAGE_RATE = .35. These are the only fields that should be declared in this class.

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

```
IPhone iPhone1 = new IPhone("123-456-7892", 20, 548, 220, 55);
```

(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.
   - `getIMessages`: Accepts no parameters and returns an int representing iMessages.
   - `setIMessages:` Accepts an int representing the iMessages, sets the field, and returns nothing.
   - `calculateBill`: Accepts no parameters and returns a double representing bill calculated as follows: invokes calculateBill in the superclass and then adds iMessages * IMESSAGE_RATE.
   - `resetBill:` Accepts no parameters, invokes resetBill in the superclass, sets iMessages to zero, and returns nothing.
   - `toString`: The toString method in the IPhone class should invoke the toString method in SmartPhone and then concatenate the iMessages information. Below is an example of the toString result for IPhone iPhone1 as it is declared above.

```
Number: 123-456-7892 (IPhone)
Bill: $95.05 for 20 Texts, 548 Talk Minutes, 220 MB of Data, 55 iMessages
```

Code and Test: As you implement the IPhone class, you should compile and test it as methods are created. Although you could use interactions, it may be more efficient to test: (1) by using the CellPhonesPart1 class and main method described below and/or (2) by creating appropriate JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of IPhone in the main method of CellPhonesPart1 and then run CellPhonesPart1 in the canvas. After you drag the instance onto the canvas, you can examine it for correctness. You should then begin creating test methods in the corresponding IPhoneTest.java file.

- **Android.java**

Requirements: Derive the Android class from the SmartPhone class.

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

(1) **Field**: an *instance* variable for hotspot minutes of type int which should be declared with the *private* access modifier; a public constant of type double for HOTSPOT_RATE = 0.75. These are the only fields that should be declared in this class.

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

```
Android android1 = new Android("123-456-7893", 500, 400, 1000, 30);
```

(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.
   - `getHotspotMin`: Accepts no parameters and returns an int representing hotspot minutes.
   - `setHotspotMin:` Accepts an int representing hotspot minutes, sets the field, and returns nothing.
   - `calculateBill`: Accepts no parameters and returns a double representing bill calculated as follows: invokes calculateBill in the superclass and then adds hotspotMin * HOTSPOT_RATE.
   - `resetBill:` Accepts no parameters, invokes resetBill in the superclass, sets hotspotMin to zero, and returns nothing.

- `toString`: The toString method in the Android class should invoke the toString method in SmartPhone and then concatenate the hotspot minutes information. Below is an example of the toString result for Android android1 as it is declared above.

```
Number: 123-456-7893 (Android)
Bill: $362.50 for 500 Texts, 400 Talk Minutes, 1000 MB of Data, 30 Hotspot Minutes
```

Code and Test: As you implement the Android class, you should compile and test it as methods are created. Although you could use interactions, it may be more efficient to test: (1) by using the CellPhonesPart1 class and main method described below and/or (2) by creating appropriate JUnit test methods. For example, as soon you have implemented and successfully compiled the constructor, you should create an instance of Android in the main method of CellPhonesPart1 and then run CellPhonesPart1 in the canvas. After you drag the instance onto the canvas, you can examine it for correctness. You should then begin creating test methods in the corresponding AndroidTest.java file.

- **CellPhonesPart1.java**

Requirements: Driver class with main method.

Design: The CellPhonesPart1 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 CellPhone and FlipPhone, you can add statements to main that create and print an instance of FlipPhone. [Since CellPhone is abstract you cannot create an instance of it.] When main is completed, it should contain statements that create and print instances of FlipPhone, SmartPhone, IPhone, and Android. Since printing the objects will not show the details of the fields, you should also run CellPhonesPart1 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 flip1, smart1, iPhone1, and android1 as described in the sections above and your main method is stopped between steps after android1 has been created, you can enter the following in interactions to calculate the bill for the Android object.

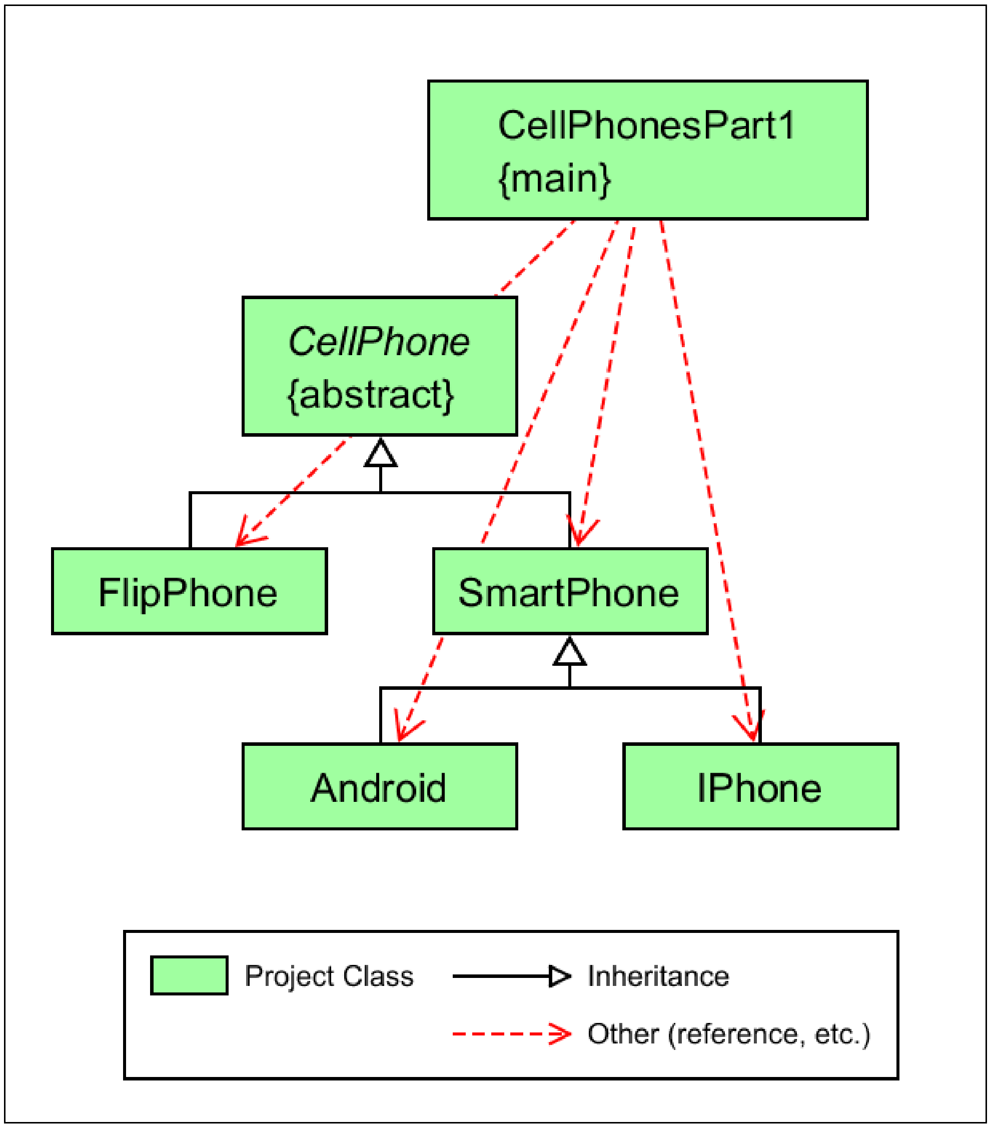
```
android1.calculateBill()
362.5
```

Code and Test: After you have implemented the CellPhonesPart1 class, you should create the test file CellPhonesPart1Test.java in the usual way. The only test method you need is one that checks the class variable *cellPhoneCount* that was declared in CellPhone and inherited by each subclass. In the test method you should reset *cellPhoneCount*, call your main method, then assert that *cellPhoneCount* is four (assuming that your main creates four objects from the CellPhone hierarchy). The following statements accomplish the test.

```
CellPhone.resetCellPhoneCount();
CellPhonesPart1.main(null);
Assert.assertEquals("CellPhone.cellPhoneCount should be 4. ",
                    4, CellPhone.getCellPhoneCount());
```

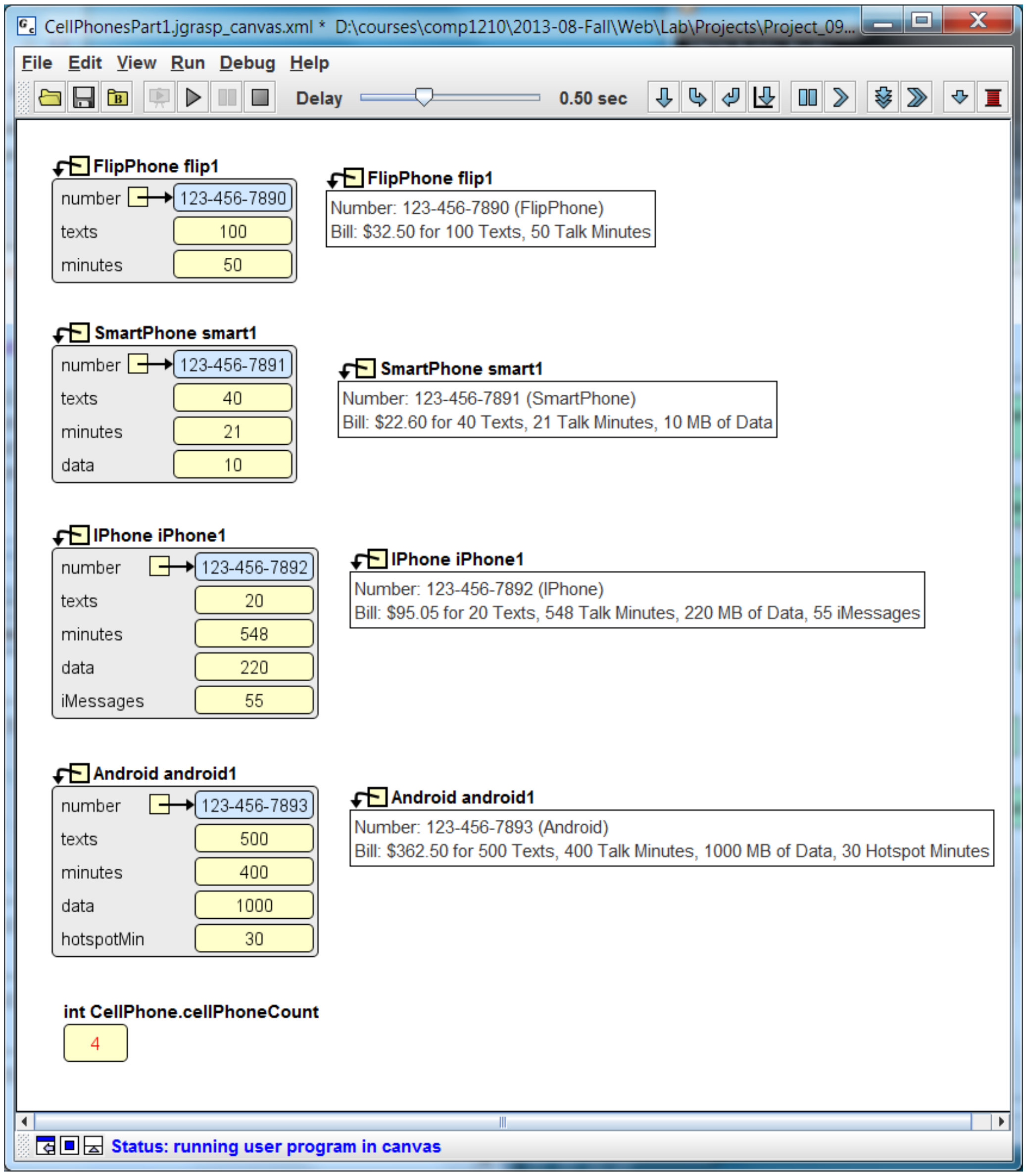
UML Class Diagram

As you add your classes to the jGRASP project, you should generate the UML class diagram for the project. To layout the UML class diagram, right-click in the UML window and then click **Layout** > **Tree Down**. Click in the background to unselect the classes. You can now select the CellPhonesPart1 class and move it around as appropriate. Below is an example. Note that the dependencies represented by the red dashed arrows indicate that CellPhonesPart1 references each of the subclasses of CellPhone (i.e., the main method in CellPhonesPart1creates instances of each subclass and prints then out).



Canvas for CellPhonesPart1

Below is an example of a jGRASP viewer canvas for CellPhonesPart1 that contains two viewers for each of the variables flip1, smart1, iPhone1, and android1. The first viewer for each is set to Basic viewer and the second is set to the toString viewer. A viewer for the class variable CellPhone.cellPhoneCount is near the bottom of the canvas. 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 *cellPhoneCount*. To display types with the labels, click **View** on the canvas top menu bar then turn on **Show Types in Viewer Labels** with the check box.


CellPhonesPart1.jgrasp_canvas.xml * D:\courses\comp1210\2013-08-Fall\Web\Lab\Projects\Project_09...
File Edit View Run Debug Help
Delay 0.50 sec

FlipPhone flip1

number	123-456-7890
texts	100
minutes	50

FlipPhone flip1

```
Number: 123-456-7890 (FlipPhone)
Bill: $32.50 for 100 Texts, 50 Talk Minutes
```

SmartPhone smart1

number	123-456-7891
texts	40
minutes	21
data	10

SmartPhone smart1

```
Number: 123-456-7891 (SmartPhone)
Bill: $22.60 for 40 Texts, 21 Talk Minutes, 10 MB of Data
```

IPhone iPhone1

number	123-456-7892
texts	20
minutes	548
data	220
iMessages	55

IPhone iPhone1

```
Number: 123-456-7892 (IPhone)
Bill: $95.05 for 20 Texts, 548 Talk Minutes, 220 MB of Data, 55 iMessages
```

Android android1

number	123-456-7893
texts	500
minutes	400
data	1000
hotspotMin	30

Android android1

```
Number: 123-456-7893 (Android)
Bill: $362.50 for 500 Texts, 400 Talk Minutes, 1000 MB of Data, 30 Hotspot Minutes
```

int CellPhone.cellPhoneCount

4

Status: running user program in canvas