Project: Cloud Storage – Part 1

Deliverables

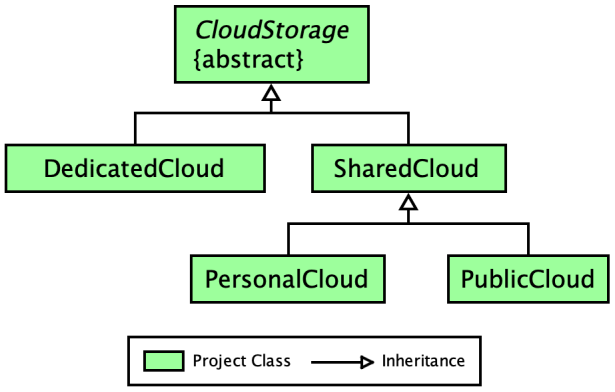
Your project files should be submitted to Web-CAT by the due date and time specified. Note that there is also an optional Skeleton Code assignment which will indicate level of coverage your tests have achieved (there is no late penalty since the skeleton code assignment is ungraded for this project).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

- CloudStorage.java
- DedicatedCloud.java, DedicatedCloudTest.java
- SharedCloud.java, SharedCloudTest.java
- PublicCloud.java, PublicCloudTest.java
- PersonalCloud.java, PersonalCloudTest.java
- (Optional) CloudStoragePart1.java, CloudStoragePart1Test.java

Specifications

Overview: This project is the first of three that will involve the monthly cost and reporting for cloud storage. You will develop Java classes that represent categories of cloud storage including dedicated cloud storage and shared cloud storage (both public and personal cloud storage). 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.

- **CloudStorage.java**

  Requirements: Create an *abstract* CloudStorage class that describes cloud storage data and provides methods to access the data.

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

  (1) **Fields**:
  instance variables (*protected*) for: (1) name of type String and (2) base storage cost of type double.
  class variable (*protected static*) for the count of CloudStorage 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 CloudStorage class must contain a constructor that accepts two parameters representing the instance variables (name and base storage cost) and then assigns them as appropriate. Since this class is abstract, the constructor will be called from the subclasses of CloudStorage 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.
  - `getBaseStorageCost`: Accepts no parameters and returns a double representing base storage cost.
  - `setBaseStorageCost:` Accepts a double representing the base storage 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 CloudStorage object. This method will be inherited by the subclasses. For an example of the toString result, see the DedicatedCloud class and SharedCloud 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 cloud storage.

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

- **DedicatedCloud.java**

Requirements: Derive the class DedicatedCloud.java from CloudStorage.

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

(1) **Field**: *instance* variable for server cost 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 DedicatedCloud class must contain a constructor that accepts three parameters representing the two instance fields in the CloudStorage class (name and base storage cost) and the one instance field for server cost declared in DedicatedCloud. Since this class is a subclass of CloudStorage, the super constructor should be called with field values for CloudStorage. The instance variable for server cost should be set with the last parameter. Below is an example of how the constructor could be used to create a DedicatedCloud object:

```
DedicatedCloud c1 = new DedicatedCloud("Cloud One", 40.00, 10.00);
```

(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.

- `getServerCost`: Accepts no parameters and returns a double representing server cost.
- `setServerCost`: Accepts a double representing the server cost, sets the field, and returns nothing.
- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the DedicatedCloud calculated as the sum of base storage cost and server cost.
- `toString`: Returns a String describing the DedicatedCloud object by calling parent's toString method, super.toString() and then appending the line for server cost. Below is an example of the toString result for DedicatedCloud c1 as it is declared above.

```
Cloud One (class DedicatedCloud) Monthly Cost: $50.00
Base Storage Cost: $40.00
Server Cost: $10.00
```

Code and Test: As you implement the DedicatedCloud 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 CloudStorage (parent class of DedicatedCloud). The test methods in DedicatedCloudTest should be used to test the methods in both CloudStorage and DedicatedCloud. Remember, a

DedicatedCloud object *is-a* CloudStorage object which means DedicatedCloud inherited the instance methods defined in CloudStorage. Therefore, you can create instances of DedicatedCloud in order to test methods of the CloudStorage class. You may also consider developing CloudStoragePart1 (page 7) in parallel with this class to aid in testing.

- **SharedCloud.java**

Requirements: Derive the class SharedCloud from CloudStorage.

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

(1) **Fields:**
instance variables (*protected*): (1) data stored 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 SharedCloud.COST_FACTOR.
These are the only fields that should be declared in this class.

(2) **Constructor**: The SharedCloud class must contain a constructor that accepts four parameters representing the two instance fields in the CloudStorage class (name and base storage cost) and the two instance fields (data limit and data stored) declared in SharedCloud. Since this class is a subclass of CloudStorage, the super constructor should be called with field values for CloudStorage. The instance variables for data limit and data stored should be set with the last two parameters. Below is an example of how the constructor could be used to create a SharedCloud object:

```
SharedCloud c2 = new SharedCloud("Cloud Two", 9.00, 12.0, 20.0);
SharedCloud c3 = new SharedCloud("Cloud Three", 9.00, 25.0, 20.0);
```

(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.

- `getDataStored`: Accepts no parameters and returns a double representing data stored in GB.
- `setDataStored`: Accepts a double representing the data stored in GB, 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.
- `getCostFactor`: Accepts no parameters and returns a double representing the cost factor. Although this method is returning a constant, it should not be *static* to ensure the cost factor for this class is returned of when called on an object of this class.
- `dataOverage`: Accepts no parameters and returns a double representing the amount data stored exceeds the data limit in GB for the cloud, calculated as (data stored - data limit), or returns zero if the value is negative.

- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the SharedCloud object as follows: (base storage cost + dataOverage() * SharedCloud.COST_FACTOR).
- `toString`: Returns a String describing the SharedCloud object by calling parent's toString method, super.toString() and then appending the lines for data stored, data limit, overage, and cost factor. Be sure to call the getCostFactor method for the cost factor value. Below is an example of the toString results for SharedCloud c2 and c3 as it is declared above. *Note that data stored, data limit, and overage should use the DecimalFormat pattern* `"0.000"`, *whereas monthly cost should use* `"$#,##0.00"`.

```
Cloud Two (class SharedCloud) Monthly Cost: $9.00
Base Storage Cost: $9.00
Data Stored: 12.000 GB
Data Limit: 20.000 GB
Overage: 0.000 GB
Cost Factor: 1.0

Cloud Three (class SharedCloud) Monthly Cost: $14.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 1.0
```

Code and Test: As you implement the SharedCloud 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 SharedCloud in a JUnit test method in the SharedCloudTest 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 CloudStoragePart1 (page 7) in parallel with this class to aid in testing.

- **PublicCloud.java**

Requirements: Derive the class PublicCloud from SharedCloud.

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

(1) **Field**: **constant (*public static final*)** COST_FACTOR of type double set to 2.0, which can be referenced as PublicCloud.COST_FACTOR outside of the class.
These are the only fields that should be declared in this class.

(2) **Constructor**: The PublicCloud class must contain a constructor that accepts four parameters representing the two instance fields in the CloudStorage class (name and base storage cost) and the two instance fields in the SharedCloud (data stored and data limit). Since this class is a subclass of CloudStorage, the super constructor should be called with all four parameters. Below is an example of how the constructor could be used to create a PublicCloud object:

```
PublicCloud c4 = new PublicCloud("Cloud Four", 9.00, 25.0, 20.0);
```

(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.

- `getCostFactor`: Accepts no parameters and returns a double representing the cost factor. Although this method is returning a constant, it should not be *static* to ensure the cost factor for this class is returned of when called on an object of this class.
- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the SharedCloud object as follows: (base storage cost + dataOverage() * PublicCloud.COST_FACTOR).
- **There is no `toString` method in this class**. When toString is invoked on an instance of PublicCloud, the toString method inherited from SharedCloud is called. Below is an example of the toString result for PublicCloud c4 as it is declared above.

```
Cloud Four (class PublicCloud) Monthly Cost: $19.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 2.0
```

Code and Test: As you implement the PublicCloud class, you should compile and test it as methods are created. For details, see **Code and Test** above for the DedicatedCloud and SharedCloud classes. You may also consider developing CloudStoragePart1 (page 7) in parallel with this class to aid in testing.

- **PersonalCloud.java**

Requirements: Derive the class PersonalCloud from class SharedCloud.

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

(1) **Field**: **constant (*public static final*)** COST_FACTOR of type double set to 3.0, which can be referenced as PersonalCloud.COST_FACTOR outside the class.
This is the only field that should be declared in this class.

(2) **Constructor**: The PersonalCloud class must contain a constructor that accepts four parameters representing the two instance fields in the CloudStorage class (name and base

storage cost) and the two instance fields in the SharedCloud (data stored and data limit). Below is an example of how the constructor could be used to create a PersonalCloud object:

```
PersonalCloud c5 = new PersonalCloud("Cloud Five", 9.00, 21.0, 20.0);
```

(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.

- `getCostFactor`: Accepts no parameters and returns a double representing the cost factor. Although this method is returning a constant, it should not be *static* to ensure the cost factor for this class is returned of when called on an object of this class.
- `monthlyCost`: Accepts no parameters and returns a double representing the monthly cost for the SharedCloud object as follows: (base storage cost + dataOverage() * PersonalCloud.COST_FACTOR).
- **There is no `toString` method in this class**. When toString is invoked on an instance of PersonalCloud, the toString method inherited from SharedCloud is called. Below is an example of the toString result for PersonalCloud c5 as it is declared above.

```
Cloud Five (class PersonalCloud) Monthly Cost: $12.00
Base Storage Cost: $9.00
Data Stored: 21.000 GB
Data Limit: 20.000 GB
Overage: 1.000 GB
Cost Factor: 3.0
```

Code and Test: As you implement the PersonalCloud class, you should compile and test it as methods are created. For details, see **Code and Test** above for the DedicatedCloud and SharedCloud classes. You may also consider developing CloudStoragePart1 (page 7) in parallel with this class to aid in testing.

- **CloudStoragePart1.java (Optional)**

Requirements: Driver class with main method is optional but you may find it helpful.

Design: The CloudStoragePart1 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 CloudStorage and DedicatedCloud, you can add statements to main that create and print an instance of DedicatedCloud. [Since CloudStorage is abstract you cannot create an instance of it.] When main is completed, it should contain statements that create and print instances of DedicatedCloud, SharedCloud, PublicCloud, and PersonalCloud. Since printing the objects will not show all of the details of the fields, you should also run CloudStoragePart1 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 c1, c2, c3, c4 and c5 as described in the sections above and your main method is stopped between steps after c4 has been created, you can enter the following in interactions to get the monthly cost and overage for the PublicCloud object.

```
► c4.monthlyCost()
  19.0
► c4.dataOverage()
  5.0
```

The output from main assuming you create print the four create c1, c2, c3, c4 and c5 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.

```
Cloud One (class DedicatedCloud) Monthly Cost: $50.00
Base Storage Cost: $40.00
Server Cost: $10.00

Cloud Two (class SharedCloud) Monthly Cost: $9.00
Base Storage Cost: $9.00
Data Stored: 12.000 GB
Data Limit: 20.000 GB
Overage: 0.000 GB
Cost Factor: 1.0

Cloud Three (class SharedCloud) Monthly Cost: $14.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 1.0

Cloud Four (class PublicCloud) Monthly Cost: $19.00
Base Storage Cost: $9.00
Data Stored: 25.000 GB
Data Limit: 20.000 GB
Overage: 5.000 GB
Cost Factor: 2.0

Cloud Five (class PersonalCloud) Monthly Cost: $12.00
Base Storage Cost: $9.00
Data Stored: 21.000 GB
Data Limit: 20.000 GB
Overage: 1.000 GB
Cost Factor: 3.0
```

Code and Test: After you have implemented the CloudStoragePart1 class, you should create the test file CloudStoragePart1Test.java in the usual way. The only test method you need is one that checks the class variable *count* that was declared in CloudStorage and inherited by each subclass. In the test method, you should reset *count*, call your main method, then assert that *count* is five (assuming that your main creates five objects from the CloudStorage hierarchy). The following statements accomplish the test.

```
CloudStorage.resetCount();
CloudStoragePart1.main(null);
Assert.assertEquals("CloudStorage count should be 5. ",
                    5, CloudStorage.getCount());
```

Canvas for CloudStoragePart1

Below is an example of a jGRASP viewer canvas for CloudStoragePart1 that contains a viewer for the class variable CloudStorage.count and two viewers for each of c1, c2, c3, c4 and c5. 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*.