Lab 3: Class Design Part II

Composition, Encapsulation, Overloading, Constructors, Pass-By-Reference, Static, Final, Getters/Setters

CSSSKL162: Programming Methodology Skills

Summary

In this lab we will design even more classes, using techniques we've covered in class. In each section, we'll practice basic class design, ***encapsulation***, ***overriding* & *overloading*** methods, building ***constructor*** definitions (including ***copy constructors***), adding ***access modifiers*** to variables and methods, using ***getters/setters***, and finally working with variables across method boundaries (i.e., understanding ***pass-by-reference*** vs ***pass-by-value***).

Classes As Contracts

When we build a class to match a certain interface, we're doing more than just "following the homework or instructor's directions". We're agreeing to a contract of sorts, promising to have specific methods that accomplish specific tasks. We could define multiple class interfaces (think radio faceplate or front-end here) as we have done in this lab, and multiple teams working in parallel could develop them. We'll practice this paradigm by reading about a class's methods and data items it should have, then implementing those methods and data members ourselves. This is like me showing you an outline of a car we're going to design, and you (the designer) filling in the outline with details about what it means to be a car. What can I do? These actions will be defined as public methods in your interface. What am I composed of? These will be realized as instance variables defined in the Car class. Lets start by reading the brief descriptions of the classes below, and then designing each class to include the indicated methods and data – think of this part as you fulfilling a contract each time you build a class to a specific interface (interface: set of methods). Sometimes, when fulfilling a specified interface (like in the classes below and your second Fraction assignment), it may help conceptually to declare your data items first (what a class *has*) and

then the methods that work on the defined data (what a class *does*); in this way, "Data structures programs".

The Date Class

This class will function similarly in spirit to the Date class in the text, and behaves very much like a "standard" class should. It should store a month, day, and year, in addition to providing useful functions like date reporting (toString()), date setting (constructors and getters/setters), as well as some basic error detection (for example, all month values should be between 1-12, if represented as an integer). Start this by defining a new class called "Date", and implement the data items and methods listed in your "contract" below. By declaring both data and methods that will operate on this data in one file, we are "wrapping up" chunks of dedicated functionality (methods) with the data items that they operate on in one clean and simple abstraction: the Class.

Class Invariants

Enforce the following "rules" that your class should never violate. By doing this you maintain the coherency of your object's internal state – this is also known as "never invalidating the state (data) of your object". In our **Fractions** homework, this would involve modifying getters/setters so as to never accept a zero denominator. Here we'll consider some Invariants for the **Date** class.

- All month values should be in between 1-12
- All day values should be in between 1-31
- All year values should be positive.

Data Members

- Declare a variable for month, day, and year.
  - Define these as private? public?
    1. What is the difference between the public and private access modifiers?
    2. When data is defined as static…
       a. Can it be accessed or read?
       b. Can it be written to?
       c. If we had declared one static variable and 10 objects declared in RAM, how many static variables would also be in memory?

3. When data is defined as final…
    a. Can it be accessed/used or read?
    b. Can it be written to other than the first initialization?
    c. Why would it be ok to declare a final (or static final) variable as public?
    d. Later: How does the concept of a final variable relate to **Immutable** classes?

Method Members

- Provide getter/setter methods for each of the variables above
- Provide logic in your setter methods to observe the following Class Invariants
- public Date() {}
    - This is the default no-arg constructor.
- public Date(int m, int d, int y);
    - Should we use month = m; or setMonth(m)? What are the differences?
- public Date(Date other);
    - This method is a copy constructor; initialize your object's data members (this.month, etc) using "other".
- public String toString(); //report on your date
- public boolean equals(Object other);
    - In this method, you'll compare "this" to "that", once the other object has been checked and cast.
    - See Fraction equals() for proper null-check and casting.

Sample Output

a is :0.0.0
b is :2.1.2030
c is :2.1.2030
B and A are equal: false
B and C are equal: true

The Shape Class

When designing your **Square** and **Circle** classes from last week, it would have been useful to have a template to start from, rather than starting from scratch. Let's define such a template here, a class that will contain data items common to all Shapes (such as an X, Y, or a Color – all shapes have these elements). We'll include a few methods that won't be used by this labDriver, but don't worry…these may prove useful in the future. We'll be satisfied here with designing yet another class, even if some of the methods are just stubs for now. For our Paint program as it stands currently, all shapes will have the following data items and methods…

Class Invariants

- Declare 4 class invariants for the Shape class as comments and submit them.
    - What might be a good invariant? Think about how we might use this Shape…
        - Perhaps x and y must be nonnegative integers?
        - TODO: declare invariants for this Shape class here and in comments.

Data Fields

- Declare an x and a y. (all data here is private)
- Declare a Color object.

Instance Methods

- public Shape();
- public Shape(int x, int y, Color color); //remember imports for class Color
- public Shape(Shape other); //copy c'tor
- public String toString(); //describe your shape's x,y, color, etc.
- public double getArea(); //to be replaced by subclasses, return -1 here
- public void draw(Graphics g); //to be used by the paint program, empty for now

Sample Output

a: Shape (120,0) and color:null and area: -1
b: Shape (10,140) and color:java.awt.Color[r=64,g=64,b=64] and area: -1
c: Shape (10,10) and color:java.awt.Color[r=64,g=64,b=64] and area: -1

The CharList Class (A.k.a String or StringBuffer Class)

Some classes are unusual in that once created & initialized via a constructor, they never change their internal data for the lifetime of the object. The term mutable (think editable) is used to describe classes whose internal state (data) can change as the object is used. An ***Immutable*** Object is an object that once defined, never alters or changes its internal data – the java.lang.**String** class is one example of this. ***Immutability*** is like a "read-only" characteristic; the data may be observed and "gotten" but is never changed or "written to". We'll discuss ***privacy leaks*** shortly in class, but it's worth noting here that since an Immutable object can't be changed, we don't have to worry about privacy leaks across getter/setter boundaries when those that do the getting/setting cannot change your object. Think of this as similar to a "public static" data item – we don't worry about making it private, since there is no damage an external client can do to these data items. We'll build a String class here that is really just a **CharList** or **IntList** variation that we've seen before. Start by defining a new class called "**CharList**" and implement the data and method members below

Class Invariants

- Ask the instructor if you may assume no string will grow beyond 100 characters in this lab.

Data

- An array to store characters.
- Should we make this data private?
- An integer variable used to track the number of characters

Methods

- public CharList() {}
- public CharList(String startStr); //use string to set up internals
- public CharList(CharList other); //copy c'tor; be sure to check for null input
- public void add(char next); //Could we make this so it dynamically grows to accommodate more than 100 elements?
- public char get(int index);
- public String toString();
    - Return a string that is the concatenated result of combining every character in your char array.

- public boolean equals(Object other) {
    - if(other == null || ! (other instanceof CharList)) return false; //follow this pattern to check for null and verify class types
    - CharList that = (CharList) other; //use this v.s. that from this point on
        - Two strings are the same if they share the same length and the same characters

Sample Output

```
a is :katnis and has 6 chars
b is :Batman and has 6 chars
c is :Batman and has 6 chars
B and A are equal : false
B and C are equal : true
```

The LineSegment Class – Class Composition ("Has a") & Privacy Leaks

Find your Point2D.java class that you built in the previous lab. The **Point2D** should store an x and y coordinate pair, and will be used to build a new class via **class composition.** A **Point2D** *has a* x and a y, while a **LineSegment** has a start point and an end point (both of which are represented as **Point2Ds**). This is similar in spirit to your next homework, where we'll build a few simple classes called **Money** and **Date**, and then build a meta-class or container class that is composed of both a **Money** and **Date** object – we'll call this a **Bill** object, and we can state that a **Bill** *has a* **Money** and *has a* **Date** object inside of it. Note that when a class offers getters/setters for a primitive, pass-by-value ensures that changes to copies of a private primitive won't affect the original primitive. When we pass objects to and from methods (as we do with getters and setters), objects are shared due to pass-by-value. This results in a *privacy leak:* objects you marked as private are still directly accessible due to the memory semantics involved with pass-by-reference. As a result, when we get and set objects, extra care in the form of cloning objects is required to avoid such privacy leaks. In the end, we'll copy our objects and emulate pass-by-value with our objects so that people who try to "get" our private objects actually get a clone of the object – if they destroy the clone, your private state objects will not be affected. Pay special attention to the getters/setters associated with your start and end points; notice how eclipse incorrectly writes this code, too.

Class Invariants

- The start and end points of a line segment should never be null
    - Initialize these to the origin instead.

Data

- A **LineSegment** *has a* start point
    - This is a **Point2D** object
    - All data will be private
- A **LineSegment** also *has a*n end point.
    - Also a **Point2D** object

Methods

- Create getters and setters for your start and end points
    - public Point2D getStartPoint() {
    - public void setStartPoint(Point2D start) {
- Create a toString() function to build a string composed of the startPoint's toString() and endPoint's toString()
    - Should look like "Line start(0,0) and end(1,1)"
- Create an equals method that determines if two LineSegments are equal
    - public boolean equals(Object other) {
        - if(other == null || !(other instanceof null)) return false; //use this as the first line
        - LineSegment that = (LineSegment) other; //after this line, use this v.s. that
        - //return start and end points are equal, requires an equals in the **Point2D** class
- Uncomment the method call in main to invoke the driver associated with the **LineSegment** code.
    - Fix each error as you encounter them in the driver for **LineSegment**
- Create a default, no-arg constructor
    - This should define the start point and the end point to be at the origin
- Create an overloaded constructor that takes a start point and an end point
    - This should check for null's for the start and end point

- Create a copy constructor (also overloaded) that takes a **LineSegment** object and initializes this using other.
    - public LineSegment(LineSegment other) {
- Create a distance() function that will calculate the line distance using the distance formula
    - Hint: Math.sqrt(), Math.abs()
- Run the driver code in ClassDesignII.java to test your **LineSegment** class.
- Answer the following questions as comments in your **LineSegment** code:
    - What is a privacy leak?
    - Do your getters or setters have privacy leaks?
    - Where else could a privacy leak occur?

Sample Output

```
Line a: LineSegment [startPoint=Point2D [x=3, y=3], endPoint=Point2D [x=4, y=4]]
Line b: LineSegment [startPoint=Point2D [x=1, y=1], endPoint=Point2D [x=2, y=2]]
Line c: LineSegment [startPoint=Point2D [x=1, y=1], endPoint=Point2D [x=2, y=2]]
Line b's distance between points: -1.0
Does a equal b? false
Does a equal c? false
Does b equal c? true
```

The Fraction Class – Immutable Classes

In this section, we'll build another Fraction class that is unchangeable once initialized and uses the keyword **final** for its numerator and denominator. Such a Fraction object will have all of its data declared **final,** and is our first example of building an immutable class. Once a Fraction object is built, its data items will never change for the lifetime of the object. Another way to view this is that the object is completely **read-only**. As a result, its data will also be declared **public,** which is the only example of public data you'll find in this quarter. If you wish to change a fraction object's numerator or denominator, the old object must be discarded and a new object created in its place. Again, this object's data will be immutable, constant, non-variable, unchangeable, non-editable, or read-only. So, to add two fractions, our add function will return a new Fraction object that is the sum of the two previous (unchangeable) fractions we wish to add. Once you've built this class, uncomment out the appropriate tests in the driver for this lab. Start by building a new Fraction class, and define the following members:

Class Invariants

- Numerators and denominators are unchangeable once set by the constructor.
- No denominator will be stored as a 0. (i.e., no DivideByZero Exceptions).
- A Fraction is always in reduced form (reduce in the constructor to ensure this).

Data

- Define a numerator that is **public** and **final**.
  - Why don't we make this data private?
- Define a denominator that is **public** and **final.**
  - What data types should these items be?

Methods

- Define a constructor that takes a numerator and a denominator
  - Do not define a no-argument constructor. Why?
- Define a constructor that takes a Fraction object and makes a copy of it.
  - public Fraction(Fraction other){…}
- Define a toString() function as we've done for other classes.
- Define an add function that takes a fraction, adds it to this, then returns a new Fraction object that is the result of the addition of the two
  - public Fraction add(Fraction that) {//add this and that together; remember to consider the denominator here!
- Define an equals(Object o) function that has the form:

```
public boolean equals(Object other) {
        if( other != null && ! (other instanceof Fraction ) ) return false;
            //what does this code do?
        Fraction that = (Fraction) other;
            //and this code?
        //todo: code goes here
}
```

Sample Output

```
a:1\2
b:3\4
c:3\4
a:10\8
```

b:3\4
c:3\4
a.equals(b):false
b.equals(c):true

The Math2 Class – Static Classes

Other types of unusual classes include so-called "utility" classes, which are unique in that you don't need to make an object of the class to use methods defined in the class. **Math** is a good example here; we don't make a new **Math** object to use min() or random() – we just call the function with the class name to the left of the dot (**Math**.min()) rather than the object variable name. **JOptionPane**.showMessageDialog() or **Integer**.parseInt(**String**) are other examples of helper (static) methods that need no object target to execute. We can even go so far as to completely block instantiation of such a class by defining one constructor and making it private. Note how a method must be declared "static" for you to be able to call it without an object target. If a static method wishes to reference non-local variables, those data items must also be declared static. In this context, we can make the most sense of a method marked static by thinking of it in terms of "no target object" and thus no "this" implicit reference. If we have no "this" reference, we also cannot access any instance variables (ie, any variable whose full name would be "this.x", "this.numerator", etc.). The methods defined in the **Math2** class should be declared static; data like PI or E should also be declared as static or "owned by no object – shared amongst all objects".

Class Invariants

- All methods & data must be static.

Data Items

- Define a static constant (use **final**) for the mathematical constant pi.
- Define a second constant for E.
- Should we make this data private? public?
- Note that we avoid calling this section Data Members, as these are not instance variables but rather static, shared data.

Methods

- public int max(int a, int b);
- public double max(double a, double b);
- public int pow(int base, int exp);

Sample Output

The larger of the two is 20.0
And the larger of the two is 10.34
2^8 is 256

Final Section: *Pass-By-Reference & Pass-By-Value Demo in the ClassDesignIIDriver Class*

Look inside the driver class to find the passByReferenceDemo() function.

1. Uncomment this function call in main
2. Trace its execution by adding a debugging **breakpoint** in the gutter area of your text editor and starting your program in debug mode.
3. On observing the execution of the demo code, answer the following questions…
    a. What is the primary difference between passing a primitive to a method versus passing an object?
    b. When a primitive is passed to a method, does the scope of that primitive change (i.e., grow to include the called method)?
    c. What about when an object is passed to a method?
    d. How then would you describe the scope of an object that has been passed to a method?

Sample Output

a is :3
b is :java.awt.Dimension[width=0,height=0]
a is :3
b is :java.awt.Dimension[width=-1000,height=-1000]