Lab 2: Basic Classes & Arrays of Objects

CSSSKL 162: Programming Methodology Skills[1]

Summary

Build a set of small and simple classes to represent a **Date**, a **Point** on a 2D map, and a list of integers such as {3,5,-2}. Once you've practiced building smaller classes, we'll build a few larger classes that share much in common – the **Circle** and the **Square**. We'll end our class design by taking our **IntList** class and changing it to an **ObjectList** (or Picture) class that can store any object types, such as {**Circle**, **Square**, **String**}. On completion, we'll have practiced the following concepts with our code: **getters/setters**, **access modifiers**, **overloading constructors**, **equality,** and **overriding** toString().

Introduction

This lab is meant to serve as a reintroduction to Java programming, using familiar constructs such as **primitives**, **arrays**, and instantiating **Objects** from custom **Classes** we build here**.** By building user-defined classes, we are also taking our first steps towards understanding **Object-Oriented Programming**. Note that we will not use inheritance directly for this project, but once we've covered more regarding this technique, we will certainly revisit and improve this code. In the lab below, answer all questions in the form of comments inside any of your code files before you submit them.

Test Your Code as You Progress

Test the classes you will write as you build each class; *start by downloading the* **ShapesPictureDriver.java** *file from the Canvas, uncommenting the appropriate tests, and running its main* with your classes also in the same directory. Also, take a peek at the final output on the last page of this lab to see what we're trying to accomplish. Note that you will have to build a main() driver for at least one of the first few simple classes below – use the example main() driver function provided for **Point2D** as a template.

Class Design – Date V0.0 & V1.0

Create a new project for this lab and create a new class inside called **Date**.

[1] This lab is designed by R. Nash; edits by A. Retik

- What does it mean to be a **Date**?
    - What data items (called "state") do you need to keep track of?
        - Alternatively, what does a date have? ("has a" == composition)
- What can a **Date** do?
    - What are the actions and verbs that can be applied to a date? These become the class's methods.
- Try adding 3 data items to your **Date** to manage the month, day and year.
    - Should these be local variables? Class-level (or instance) variables?
- Now let's build a method to set the date
    - This function should take 3 integers as input and assign these to your instance variables declared in the previous step.
        - public void setDate(int m, int d, int y) {
- And, let's build a method to report the date
    - This function takes no input and uses the console to output the **Date** in the format "mm/dd/yyyy"
        - public void report() { //prints out "10/21/2015"
- Create a simple **main()** to test your **Date** in class.

Class Design - Point2D

Let's start again by defining a small, simple class. This class will store two integers: x, and y. This class will be able to set each of the coordinates, translate (a.k.a move or reposition) itself, and be able to reset itself to the origin. As such, we'll need to define two ***data members*** to track the x and y-axes, and a few functions to accomplish this behavior. Start by defining a new class called **Point2D** and copying the main driver code included below. Then, add each of the following functions and data items indicated in the member sections.

Data Members

- Declare two class-level variables (or fields) called x and y.
    - Make them private ints.
        - What effect does "private" have for methods trying to use x or y inside this class file?
        - What about methods trying to use x and y outside this class?
        - Can you use x and y without an associated object (or instance)?

Method Members

- public void setX(int nX) {
    - This method should set your private variable x equal to nX.
- Public void setY(int nY) {
    - Similar to above but for y
- public int getX() {
    - This function should return a copy of your private integer x;

- public int getY() {
  - Similar to above, but for y.
- public void resetToOrigin() {
  - This function sets both x and y to zero.
- public void translate(int dx, int dy) {
  - This method adds dx to x, and dy to y

-
  ```
  @Override        //Question: what does @Override do?
  public String toString() {
          return ?;  //returns a string representation of the point
  }
  ```

-
  ```
  //@Override not used here on purpose
  public boolean equals(Point2D that) {
          //compare this vs that

          return ?;  //returns true if this is equal to that; don't just use "this == that"
  }
  ```

-
  ```
  //sample driver to use as a starting point; add more code below
  public static void main(String[] args) {

          Point2D a = new Point2D();
          a.setX(5);
          a.setY(2);
          System.out.println("Point at (" + a.getX() + ", " + a.getY() );
          a.translate(-1,-1);
          System.out.println("Point at (" + a.getX() + ", " + a.getY() );
          a.resetToOrigin();
          System.out.println("Point at (" + a.getX() + ", " + a.getY() );
          Point2D b = new Point2D();
          Point2D c = new Point2D();
          System.out.println(b.toString());
          System.out.println(c);  //Question: why don't I need c.toString() here?
          System.out.println("Are b and c equal:" + b.equals(c));
  }
  ```

Class Design – IntList (A.k.a. ArrayList (v0.0))

Let's build a useful class that wraps up an array of integers. This class will store two data items; one an array of integers, and the other a counter to track the number of actual elements in the list. Start by defining a new class called **IntList** and copying the main driver code included below. Then, add each of the following functions and data items indicated in the member sections.

Data Members

- Declare an integer array called “data”
    - Make this private and able to store up to 100 integers
- Declare an integer called numElements
    - Make this private, and set it to zero.

Method Members

- public void add(int n) {
    - this should add the value n to our array and add one to num_elements
- public String toString() {
    - String retVal=””;
        - In a loop, append all the elements in our data array to this string, separated by a comma
    - return retVal;
- Copy the main driver below and run it to test your **IntList** so far
    - Do you see output for each integer added to your list?
    - “95 100 58 ” //for example
- Next, uncomment each of the commented lines in main and build the corresponding function
    - Build a function to sum the integer elements stored in your **IntList**.
    - public int sum() {
        - This function takes no parameters; uses the class-level data[] array instead
- Since the next function is similar to sum(), copy and paste sum() and rename it to “indexOf”
    - Also, add an input parameter to the method signature called target, which is an int that we’re looking for in our list of integers.
    - public int indexOf(int target) {
        - takes a target parameter that we’re looking for
        - This returns -1 if not found
- (Optional) Add a save() function inside this class so that it writes to disk all of the elements in the list.

```
public static void main(String[] args) {
        IntList a = new IntList();
        a.add(95);
        a.add(100);
        a.add(58);
        System.out.println(a.toString() );
        //System.out.println(a.sum() );
        //System.out.println(a.indexOf(100));  //uncomment these to work on next
        //System.out.println(a.indexOf(20));
        //System.out.println(a.save() );
}
```

Deeper Class Design - the Square Class

In this section, your job will be to write a new class that will adhere to the methods and data outlined in the "members" section below (said another way, your classes will *implement* a specific set of functions (called an *interface)* that we'll describe contractually, ahead of time). These classes will all be shapes, with actions (methods) that are appropriate to shape objects, such as "getArea()", or "draw()". First we build a **Square** class that contains the following data and methods listed below, and then copy it to quickly make a **Circle**.

Note: when considering each data item below, think about what *type* of data would best represent the concept we're trying to model in our software. For example, all shapes will have a coordinate pair indicating their location in a Cartesian coordinate system. This x and y pair may be modeled as ints or floats, and you might choose between the two depending on what the client application will do (i.e., how it would use these shapes and at what precision).

Data Members (the state of our object; these are *private class-level* variables):

- Define an x position variable.
  - What's a good primitive type for this?
- Define y position variable.
  - Make this private and the same type used for x.
- Define a variable to represent the sideLength amount.
  - Use a double for this.
- Add a String to represent this shape for console output (for now, just "[ ]").
- A **Color** (optional) (use java.awt.**Color**)

Public methods (the object's actions; often called its *interface)*:

- A **Square** constructor that takes no arguments
  - public Square() {
- A **Square** constructor that takes an initial x,y pair
  - public Square(int nx, …) {
    - Note that we're *overloading* this constructor.

- A **Square** constructor that takes an x,y pair and a starting side length.
    - Another overloaded constructor.
- Build a draw() method that outputs to the console the characters used to represent this shape ("[]")
    - public void draw() {
- Next, let's build our accessor methods:
    - int getX()
        - This returns the value stored in x.
    - int getY()
        - Accessor method that returns the value of y.
    - double getSideLength()
        - Another accessor method for getting the value of the **Square's** side length
    - double getArea()
        - returns the calculated area for this **Square**
- Now, our mutator methods:
    - void setX(int nX) { //changes x to nX
    - void setY(int nY) { //changes y to nY
    - void setSideLength(int sl) { //change sidelength's value to sl
- Build a reporting method called toString() that returns the characters associated with that shape.
    - @Override
      public String toString() { //sample : "[]"
- Build an equals function to determine if two Squares are the same
        - public boolean equals(Square that) {
            - Returns true if the x,y pairs match & the sideLengths match.
            - (hint: this.x == that.x && …)

The Circle Class

Next, create a **Circle** class that will represent a different shape abstraction. Note that a **Circle's** area is calculated differently than a **Square's**, and so we can expect methods to behave differently when we compare the two classes. Also notice that the differences aren't limited to methods, as a **Circle** introduces a new data item (a radius) that **Squares** lack. These classes are truly different, but yet certain features seem shared amongst the two: A common {x,y} pair, and a common set of operations such as draw() or getArea(). ***It will be quicker to cut-and-paste*** from your Square.java into a new file (Circle.java) and make a few changes, rather than starting from scratch.

1. Start by creating a new class called **Circle**.java and copy over everything from **Square**.java.
2. Change the class definition name from "public class **Square**" to "public class **Circle**"
3. Change the "double sideLength" instance variable to "double radius"
    a. Change the getSideLength() and setSideLength() to get and setRadius() instead.
        i. In eclipse, right-click on the definition of the instance variable sideLength
            1. Choose "refactor"->"rename"
                a. Rename this variable "radius"
4. Change the textual representation from a "[]" to a "O"
5. Change the calculation in getArea() from a **Square** (side*side) to that of a **Circle** (PI*r^2).
6. If you built an equals in **Square**, change the code that checks sidelengths ("this.sideLength == that.sideLength") to check radii instead, as in: "this.radius == that.radius".
    a. If you performed the refactorization step above, this was done for you automatically.
7. While building this second class, think about how you might leverage the similarities of these two different classes (Square and Circle) without having to resort to cutting and pasting.

Data Members

- An x position, stored as an integer, just like in **Square**
- A y position, also stored as an integer, just like in **Square**
- A radius, unique to **Circle**
    - We will use a double for this value.
- A String representation for console output ("O")
- (optional) A **Color** (use java.awt.**Color**)

Method Members

- A **Circle** constructor that takes no arguments (no-arg, default constructor)
- A **Circle** constructor that takes an initial x,y pair
- A **Circle** constructor that takes an x,y pair and a radius
- void draw() – Outputs to the console the character used to represent this shape ("O")
- int getX() – accessor
- int getY() – accessor

- double getArea() – accessor that calculates and returns the area for this shape
- void setX(int) – mutator
- void setY(int) – mutator
- void setRadius(double) - mutator
- Override toString() – Print the character(s) associated with this shape ("O")
- (optional) public boolean equals(Circle that)
    - Compares this and that, returns false if they are not equal
    - Returns true if the x,y pairs match && the radii match

The Bigger Picture – The ObjectList (or ArrayList (v1.0)) Class

A picture is simply a composition of shapes, and in this last section, we'll build a class used to manage such a picture. We'll create this new class by reusing code from an existing piece of software. We'll create an **ObjectList** class that will contain, amongst other state items (data), an array (or list) of Objects that are the **Squares** and **Circles** in the Picture to be drawn. **ObjectLists** will be a simple abstraction here, and will just "draw" shapes to the console in the order that they appear in the list –ignoring the coordinate pairs stored in each Shape for now. Again, note the static storage restriction of only 100 shapes per picture; in future sections, we'll learn how to dynamically resize our arrays. Aside: when we get to working with any of the Java graphics framework classes and/or swing, then a simple use of this **ObjectList** class (called a **PicturePanel** extending **JPanel**) would accommodate for much more interesting shape behaviors and interactions.

1. Copy-and-paste the **IntList**.java code into a new file called **ObjectList**.java
    a. Note this is the same process we used to quickly create the Circle.java class
2. Change the class name from **IntList** to **ObjectList**.
3. Change the instance variable that is your int[] array to an Object[] array called myShapes.
4. Change your "public void add(int nx)" method to "public void add(Object nx)".
5. Remove functions sum(), indexOf(), and save() and recompile using the provided ShapesPictureDriver.java driver.
6. Run your main and notice it still processes a list of integers correctly
7. Now run the driver code and uncomment the ObjectList segment and execute it.

Data Members

- private Object[] myShapes; //used to be "data[]"
    - This is an array of type Object, which can thus store any object of any class
        - What about primitives?
- private int numElements;
    - This integer tracks the number of live shapes in our array

Method Members

- void add (Object shape);
  - This function adds the Circle or Square to the array of circles
- @Override
  public String toString(); {
  //iterates through the array, calling toString() on each shape and appending this
  //to one large string to be returned

Sample Output

```
[] [] O O O
```

Alternate Sample Output

```
[]

[]

O

O

O
```

Questions & Observations

Answer the following questions as multi-line comments in your code…

(1) Why did we do so much copying-and-pasting in our software above?
   a. How can this approach be problematic?

(2) Are there obvious improvements that could be made here with respect the software design for Squares and Circles?

(3) What programming constructs were you familiar with, and which did you need to look up?

(4) Assume we used a separate array for **Squares** and for **Circles** rather than one unifying **Object** array.
   a. How would this complicate the task of adding a new Shape (say, a **Triangle**) to our **ObjectList** class?