COP5339 Object Oriented Software Design

Homework 3

Instructor: Dr. Ionut Cardei

The following problems are from Chapters 3 and 4.

Instructions for Organizing Your Code

1. If you use Netbeans (recommended):
   - create a new project called HomeworkX (where X is the homework #, 1 or 2,...).
   For each programming problem create a new package in the project view (the tree view on the left side of the window). Package names should be q1, q2,... for each question/problem.
   Create or edit classes in the corresponding package for each problem.

2. If you write programs with a text editor and compile and run programs from the command line:
   - create a new directory Hi, where i is the current homework number (1,2,...).
   - for each programming problem j create a separate subdirectory Hi/qj in directory Hi, where you place all files belonging to problem j (e.g. directory H1/q3 stores the files for problem 3 in homework 1).

Preparation/Delivery Instructions:

1. Write all your answers, in the order given in the homework file, in ONE PDF file.
Follow this format.

- Write your name followed by the section number (e.g. COP5339 001). For each problem write as a heading the problem number (e.g. "4.1"). The problem number must be clearly readable before the problem solution.
- Java source files must be properly identified: write the file name as a heading, followed by the file content.
- Make sure Java code and UML diagrams are readable. Nice color syntax highlighting is not required, but appreciated by the graders. Good option: http://hilite.me
- Proper indentation and code formatting are required.

2. Upload the PDF file to Canvas.

3. Upload your source .java files to Canvas.

4. Do **not** include files generated by javadoc in your submission.

5. For full credit, your designs and code must follow the course guidelines and must compile without warnings and work correctly, as required in the problem description.

Reminder: it is academic misconduct to submit work that is not your own.
Do not use code taken from the web or your colleagues.

However, you may use part of a solution any code from the textbook for your answers, e.g. the Greeter classes from Chapter 1.

Other general advice that will help you do well in this class. And build better code, too.

- !! Ask your instructor if you have any questions about the homework (and anything else related to the class) !!
- Consult the solutions for selected textbook problems, available at
  http://www.horstmann.com/oodp2/solutions/solutions.html
- Do exactly what the problem asks you to do. There is no extra credit for unnecessary work. Points are deducted if design or implementation requirements are not met.
- Do not rename classes and methods if they are given.
- Do not change method signature, where specified.
- Design/code your classes for general use. Assume there are other programmers who will use your code.
- Avoid unnecessary side effects. Do not use static fields/methods, unless warranted, e.g. main(), utility functions that don't need an implicit parameter object (this), or when you need to share a variable between class instances, or for constants.
- Check for errors and exceptions.
- Consider enclosing methods that may throw exceptions within a try-catch block.
- Do not let checked exceptions outside the *main* function.
- Check parameters and variables before you do something in a method.
  - E.g. `average = sum / list.size();`
    may throw an ArithmeticException (divide by 0) if the list is empty.
- Do not define instance variables when local variables could do the job.
- Use nouns for class names and verbs for method names.
- Follow coding conventions; class names start with capitals, methods and variables start with lowercase, etc.

Scoring: 100 points + 5 points extra credit

Chapter 3

3.1. Answer these questions:
a. Explain why encapsulation is an important principle for OO design.
b. When is it OK to throw exceptions as part of the contract ? Add to your answer an example (no code, just a description).
c. Explain why side effects should be avoided.
d. Give an example (from a real API or imagine one) where Cohesion is in conflict with Completeness and Convenience. Do not use the textbook examples.

3.2.
a) Write a class for representing complex numbers, called math.Complex.
Package name is 'math'.

The real and imaginary parts must be of type *double*.
Use http://en.wikipedia.org/wiki/Complex_numbers as a reference.

IMPORTANT: class Complex MUST BE IMMUTABLE.

The Complex class should have the following public interface:

- constructor taking real and imaginary part
- constructor taking only the real part (imaginary defaults to 0)
- method toString() that returns the string "3 + 2i" for a Complex object Complex(3,2) and for Complex(3, -2), the string "3 - 2i"
- accessors r() for the real part and i() for the imaginary
- method add() for adding two Complex numbers, returning a new Complex object, z.add(w) returns z + w
- method sub() for subtracting two Complex numbers, returning a new Complex object: z.sub(w) returns z - w
- method conj() for computing the complex conjugate , returning a new Complex object.
- method mult() for computing the product of two complex numbers, returning a new Complex object: z.mult(w) returns z * w
- method div() for computing the division of two complex numbers, returning a new Complex object: z.div(w) returns z + w
- method equals() that compares two Complex numbers and returns true if they are equal.

The Complex class **must be immutable** and must show a correct interface design, as we learned in Chapter 3.
Write the contract for each method (preconditions/postconditions (@return)).
Write the class invariant.
Note that some preconditions are trivial (i.e. no preconditions), but some operations don't work if they involve division by 0.

b) Write a main() method that show how the methods above are used on same sample Complex objects.

c) Write JUnit unit test methods in file TestComplex.java for these Complex methods:

- equals
- add
- sub
- mult
- div

Make sure the tests pass with **junit**.

Note: Netbeans and Eclipse have wizards to create and run unit tests. For Netbeans, after you write the Complex class, go to Tools/Create/Update Tests.
If you don't use these IDEs, then you have to write the unit tests by hand entirely. Do the following:

Write in the test class source file, TestComplex.java.

```
import org.junit.* ;
import static org.junit.Assert.* ;
```

The, write a method testXXX() for testing each operation, like this, for add():

```
// testing Complex.add
@Test
public void testAdd() {
    System.out.println("run test add");
    double a = 1, b = 2, c = -3, d = 4;
    double e = a + c, f = b + d;
    Complex x = new Complex(a,b);
    Complex y = new Complex(c,d);

    Complex w = x.add(y);

    Complex z = new Complex(e,f);

    // set up Complex objects
    // test condition using the Complex equals() method:
    Assert.assertTrue(z.equals(w));
}
```

Use the equals() method for verification.
Make sure you test Complex.equals() before the other tests.
(@Test is an annotation defined in the junit package and must precede each test....() method.)

Chapter 4

4.1
a) Write a class Student that has the following attributes:
- name: String, the student's name ("Last, First" format)
- enrollment date (a Date object)

The Student class provides a constructor that saves the student's name and enrollment date:

```
Student(String name, Date whenEnrolled)
```

The Student class provides accessors for the name and enrollment date.
Make sure the class is immutable. Be careful with that Date field -- remember what to do when sharing mutable instance variables -- issued discussed in Chapter 3 (class Employee).

Write contracts for all methods: preconditions/postconditions. Write the class invariant in the class javadoc comment.

b) Implement a static method in class Student

```
public static Comparator<Student> getCompByName()
```

that returns a new Comparator object for Student that compares 2 Students objects by the attribute *name*.

Implement a static method in class Student:

```
public static Comparator<Student> getCompByDate()
```

that returns a new comparator object for Student that compares 2 Students objects by their enrollment date.
You must use anonymous classes that implement the Comparator interface.

c) Write a public static main() method in class Student that:
- creates an ArrayList<Student> object called students
- adds 4 new Student objects to the students list, with some made up names and dates
- sort the students list by name and display the sorted collection to System.out.
  use function getCompByName()
- sort the students list by enrollment date and display the sorted collection to System.out.
  use function getCompByDate()

4.2.
Write a program that displays a frame with the following GUI elements:
- a label that displays a simple icon with a circle initially colored solid red
- a button labeled "Green"
- a button labeled "Blue"
- a button labeled "Red"

When the user clicks a button, the circle color must change to that indicated by the button's label.

Implementation requirements:
i) when a button is clicked you need to change the icon's color, then call the label's repaint() method. This, in turn will call the icon's *paintIcon()* method, so that the icon is redrawn with the new color. In other words, you don't call paintIcon() directly, the JLabel object does it.
ii) the buttons' action listener objects must be of anonymous classes.
iii) the buttons must be created in main() and set up in a loop with loop index from 0 to 2, like this:

```
for (i=0; i<3; i++) {
    btn[i] = createButton(i,  ......);
}
```

The createButton() function you will write gets an index (0 to 2) and creates the green, blue, and red buttons, depending on the valus of i. JButton objects are returned with the proper listener attached.

Hint: Use a Color array that can be indexed by i:

```
Color[] colors = new String[]{"RED", "GREEN", "BLUE"};
```

Then, create the color with the required color:

```
final Color c = colors[i];
```

Remember, you can only refer to *effectively final* local variables from the outside scope and to instance variables inside anonymous class functions.

Extra credit 5 points:

4.3.

Consider the animation code from the end of Chapter 4. Modify the *ShapeIcon* class so that it aggregates a collection of objects implementing the *MovableShape* interface and its *paintIcon*() method takes care of painting all the given *MovableShape* objects.

Modify the animation program (*AnimationTester.main*) to display 5 cars moving horizontally, each starting from a different vertical coordinate, so that the car with index *i* moves twice as fast as the car with index *i* - 1.