AP Computer Science — Project - OfficeSupplies

ASSIGNMENT OVERVIEW

In this assignment you'll be creating a small package of files which will model various types of office supplies. The package will include at least four files: three different classes that describe items one might find in an office or office supply store, and a single tester that demonstrates how those three classes work.

This assignment is worth 50 points and is due on the *crashwhite.polytechnic.org* server at 23:59:59 on the date given in class.

BACKGROUND

Classes that model the behaviors of "real life" objects—simulations—are an important aspect of computer programming. Object-oriented programming strategies are especially useful in writing simulations: *classes* are written to represent the objects being modeled, and *methods* for those classes are written to match the behaviors of those objects.

PROGRAM SPECIFICATION

Create a package of Java classes that:

a. model a simple device, perhaps a tally counter, a pen, or a document
b. model a moderately complex device, perhaps a stapler or a coffee machine
c. model a complex device, perhaps a copy machine or a filing cabinet
d. demonstrate/test the capabilities of the three models with a test suite (`OfficeSuppliesTester.java`).

DELIVERABLES

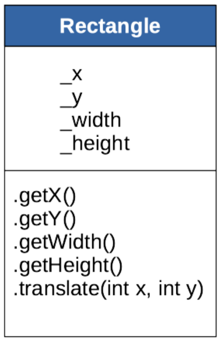
`OfficeSupplies.zip`

This single file will be a zipped directory (folder) of your project. It will include as a minimum the four files listed above (or their equivalents as designed by you), any other classes you create during the development of your program).

To submit your assignment for grading, copy your file to your directory in `/home/studentID/forInstructor/` at *crashwhite.polytechnic.org* before the deadline.

ASSIGNMENT NOTES

- This project is partly a design challenge, and partly an implementation challenge. Brainstorming what object you'd like to model, including considering the instance variables and methods that will be part of that design, is an important first step.
- To organize your thinking, draw a simple class description such as this one shown for the `Rectangle` class. You may find that you'll need to modify the class once you start implementing it, but having a diagram like this is a good way of anticipating what you'll need to code for your class.



- When writing code that simulates or models a real "thing," it's often valuable to be able to have one of those things available. Get a tally counter and play with it to see what its functions are, or get a stapler and use it to staple some documents.
- You are encouraged to get a basic working version of this package up-and-running without the addition of any *feature creep*—"Hey, we should include *this* cool thing, too!" If you wish to attempt a `RealisticStapler` that not only staples but occasionally jams up, do so only after getting a working version of `Stapler` running.
- The `OfficeSuppliesTester` will include a main program that creates instances of each of the simulations and demonstrates the features of each model. Although you might use an interactive version of the tester in your development, the final version of the tester won't require any input when run—it will simply create classes, call methods, and produce output demonstrating how those classes work.

GETTING STARTED

1. With paper and pencil, and in collaboration with your partner(s), brainstorm what objects you think you might want to write.

2. Give some thought to "real world" process of manipulating each of the Office Supplies by playing with the objects provided in class. What methods will you need to include for each of the classes?

3. Sketch out the basic features of each class, including instance variables, constructors, accessor methods, and mutator methods.

4. Consider writing some pseudocode that you can use to begin implementing those classes.

5. Create a new project that will allow you to manage this assignment. As soon as you've started to write your first class, begin writing a tester that will allow you to make sure your class and its methods are working as intended. This will save you time in the long run.

6. When your program is completed (but before the deadline), copy a final archived package (`OfficeSupplies.zip`) to the server as indicated above.

QUESTIONS FOR YOU TO CONSIDER (NOT HAND IN)

1. For many of our assignments, we create two files: a class (`TallyCounter.java`, say), and a runner or tester (`OfficeSuppliesTester.java`) that is used to work with the class. Which of those files do you like to write first? Are there advantages to writing one file before another?

SAMPLE INTERACTIONS

```
==========

TESTING TALLYCOUNTER

Creating a new TallyCounter...test passed. Counter object successfully created.

Counting a few events...test passed. Events recorded.
```

```
Checking count...test passed. Correct count recorded.

Resetting counter...test passed. Counter successfully reset.

RESULTS: 4/4 tests passed.

==========

TESTING STAPLER

Creating a new Stapler...test passed. New Stapler constructed.

Checking existence of staples...test passed. We have staples.

Stapling some things...We stapled 58 things.
Depleting the stapler...

Checking existence of staples after depletion.....
Test passed. Stapler is out of staples.

Refilling stapler...

Checking refill...test passed. Stapler refilled.

RESULTS: 6/6 tests passed.

==========

TESTING COMBINATIONLOCK

Creating a CombinationLock... successfully constructed lock.

Checking lock state... passed test. Lock is closed (locked).

Trying two numbers... passed test. Lock didn't open.

Trying four numbers... passed test. Lock didn't open.

Trying correct combination... passed test. Lock successfully opened.

Closing the lock... passed test. Lock closed and locked.

Trying to open the lock without entering new numbers...
Passed test. Lock didn't open.

Resetting combination without lock open...
Error: Lock must be unlocked to reset combination.
Error message expected above.

Trying to open with old combination... passed test. Lock successfully opened.

Resetting combination with lock open (legally)...

Closing lock with new combination entered.

Trying old combination... passed test. Old combination no longer works.

Trying new combination... passed test. New combination works.

RESULTS: 10/10 tests passed.

==========
```

EVALUATION RUBRIC (courtesy of D. Rosato)

Criteria	**Points**
The project successfully includes one object of simpler design with a small number of fields and methods, and relatively simple logic (such as a tally counter that can only increment and reset).	5
The project successfully includes one object of moderate complexity. Some methods of the object have more advanced logic such as fail states (ie., a stapler trying to be used when it is out of staples) or the object having multiple fields keeping track of different behaviors.	10
The project successfully includes one object of more significant complexity. This may include non-obvious fields or states (such as a combination lock needing to know its previously entered numbers to know if it should open) and methods requiring tricky logic (such as a copy machine knowing how many pages it will need for *x* copies of a document that is *y* single-sided pages turned into double-sided pages).	15
The project includes a tester file that showcases all abilities of the three objects designed. This includes showcasing each of the methods of the object as well as how different methods alter the state(s) of the object. The tester includes sound logic in how objects are being handled and enough *print* statements to convey to the user what is being tested and whether those tests have passed.	15
All code provided is organized and thoroughly documented. Each file should include a header comment with description, author, and version. The object's' constructor(s) and methods should each have a JavaDoc-style header comment briefly explaining the method, parameter(s), and what is being returned (if anything).	5