Due Sep 21 by 11:59pm **Points** 100 **Available** after Sep 11 at 6am

Module 2 — Methods for Simple Classes and Exceptions

Home

Modules

Assignment 1: Writing a simple class and testing it

Goals: Practice designing the representation of data, writing getter methods and testing them.

This homework will give you experience with:

- Simple class design
- Simple "composition" of classes
- Writing methods
- Writing tests
- Objects that "mutate" their values (e.g.: Employee objects change their hours over the course of time), and
- Non-mutable objects that are used to represent unchanging information (e.g. PayCheck instances that are created once for the purpose of "paying" an Employee).

1 General Instructions For This and All Future Assignments

Pay attention to naming! Our server-side test files expect your submissions to be named a certain way. Therefore, whenever the assignment specifies:

- the names of classes,
- the names and types of the fields within classes,
- the names, types and order of the arguments to the constructor, or
- filenames

...be sure that your submission uses exactly those names, spelled the same way with the exact-same capitalization, etc.

Remember to follow the directory structure we covered in class. Place your solution code in the **src** directory and your JUnit tests in the **test** directory.

1 Employees and paychecks

1.1 Introduction

Volunteer work is admirable, but many people enjoy being paid for the work they do for an employer. Depending on the type of employee, said employee may be paid by the hour, a fixed rate (salaried), or be on some bonus/commission payment scale.

This lab will focus on building two classes to manage and calculate hourly employee pay.

1.2 What to do

Write an **Employee** class that represents a single hourly employee and a **Paycheck** class that calculates employee pay for the week. Employees know their names (String), employee ID (String), the number of hours they worked in a given week (double), and their pay rate (double). Paychecks know an employee's rate, and the hours worked.
Paychecks calculate the weekly pay based as `rate * hours` if the number of hours worked is 40 or less. If the hours worked exceeds 40, the Paycheck applies an overtime rate of `1.5x` for all of the hours in excess of 40.

Create:

- An `Employee` constructor that takes a name, ID, and pay rate as parameters and sets the instance values appropriately. This constructor should also initialize the hours worked to zero (0) upon instantiation.
    - See notes below. You may assume "good" input that has been validated for all object CREATION in this assignment. Strings are non-null and numeric data is non-negative.
- An Employee method `addHoursWorked()` which takes a parameter (double) and adds the value of that parameter to the current number of hours the Employee has worked this week. Note: for this system, it is legal to add "negative" hours for the week (the employee may have checked in for the day and left early without checking out so the manager may need to do an "adjustment" on the time), however the total number of hours worked for the week cannot drop below 0.
    - We have not covered Java's exception handling mechanism yet so it is not required to be used here. However, your code should check to ensure the employee hours is always non-negative.
- An Employee method `resetHoursWorked()` that resets the employee's hours worked for the week to zero.
- An Employee "getter" method `getHoursWorked()` that returns (double) representing the hours worked for the employee.

- An Employee method getWeeklyCheck() which, when called, returns a new Paycheck object that is initialized with rate and hours worked by the Employee. Paycheck instances are immutable, so once created, their instance data should not change. We are dealing with "money" here, so it is allowable to pay employees zero if warranted. Also, we are using US dollars and we do not pay employees in fractions of cents. Round down to the nearest cent if you need to.
- A Paycheck constructor that takes the rate, and hours worked as parameters and calculates (and stores) the total pay for the week.
    - See notes below. You may assume "good" input that has been validated for all object CREATION this assignment. Numeric data is non-negative.
- A Paycheck method getTotalPay() that returns the total pay for the week.
- A Paycheck method getPayAfterTaxes() that returns the amount paid the employee after a flat tax is deducted based on this scale:
    - If the employee total payment is less than $400, deduct 10% of the total payment to obtain "payment after taxes"
    - If the employee total payment is $400 or more, deduct 15% of the total payment to obtain "payment after taxes"
- A toString method for the Paycheck class that returns a String representing the current payment AFTER taxes are assessed. The string representation should be in the form of US dollars with text as illustrated below:
  Payment after taxes: $ ###.##
- A toString method for the Employee class that returns a String, allowing Employee objects to be represented by the employee name, ID, and current week's payment after taxes are assessed.
  A visual representation of the String (if we were to print it) would be like this (note the space between the $ and the numeric value - you optionally might want to explore the java.text.DecimalFormat class to help your formatting):
  Name: Clark Kent
  ID: SUPS-111
  Payment after taxes: $ 416.50

For each method:

- Design the signature of the method.
- Write Javadoc-style comments for that method.
- Write the body for the method.

- Write one or more tests that check that the method works as specified in all cases. Include Javadoc for your test methods as well.

Notes:

As you write tests for these two simple classes: think about each test objective, and then how you will test it. Remember: one test, one objective. However it may take several assert statement to test one objective: this is perfectly OK.

We haven't covered Java exception handling yet, so you may assume "good" input data for all object CREATION for this assignment. However, you still need to write your tests to consider boundary and edge cases.

We're using US dollars here. Also note that we don't pay employees in fractions of cents either. Our company policy is that if an employee earned more than 0 but less than 1 cent, we'll pay them the 1 cent.

How to submit

1. You can upload your homework to Gradescope by create a zip file that contains directly your src and test folders. DO NOT simply drag your folders to the Gradescope upload since Gradescope does NOT preserve the directory structure unless you create a zip archive. Tip: Be sure that your src and test directories are at the ROOT of your archive. In other words, do NOT enclose src and test in a folder (e.g. HW1). Our test suite will look for the src and test directories at the top level of your submission. As a pre-flight check, unzip your .zip file before submission - when you unzip the file, you should see only the src and test folders.

   OR

2. You can submit to Gradescope via your GitHub account if you have one Be sure the branch (main or sub-branch) mirrors the directory structure we specified for the assignment. In this case, src and test should be at the top level. Go to the Gradescope submission and select GitHub as the option, then specify your homework's repository/branch.