Payroll Generator Homework Instructions

Payroll is a tedious but important task for companies. It is also a very complex task as companies grow and expand often to multiple regions in the world. Adding to the complexity is the ability to work remotely, so then systems need to be able to track a series of differences given employee type, benefits, roles, taxes, regionals laws, etc.

When computers first came out, one of their first business applications was focused on accounting and accurately calculating payroll. At the time most of the payroll systems were custom applications writing in FORTRAN due to its strength in accurate math calculations. Another popular language for early applications was COBOL due to its ability to handle large amounts of data. It was one of the first compiled languages developed by Admiral Grace Murray-Hopper's team in the 1950s.

The languages used and design of payroll applications have changed over time, but the need for accurate payroll calculations has not.

In this assignment, you will be building a simple payroll generator that reads in employee information and time cards, and then generates pay stubs for each employee.

> *[!NOTE] If you are curious about the popularity of languages, here is a nice data visualization of the most popular programming languages over time. In the end, a language is just a tool, and the most important thing is to understand the concepts behind the language. Also, there is some misleading data, as typescript is a javascript replacement, so as it grows, javascript shrinks. Same with Kotlin for Java.*

Table of Contents

Learning Objectives

- Designing with inheritance and polymorphism in java
- Practicing Test Driven Development
- Implementing a class hierarchy in Java
- Implementing junit tests for all methods
- Making use of java collections/lists
- Explore the use of streams in Java
- Practice reading and writing files in Java

Instructions

> *[!IMPORTANT] This is a two week assignment, spanning multiple modules. It is best to break up the work over those weeks, and not try to do it all in one week. We suggest week one, you focus on design, and the inheritance hierarchy, and then week two you focus on reading/writing files, the main driver method, and your final report.*

In this assignment, you will be building a simple payroll generator that reads in employee information and time cards, and then generates pay stubs for each employee.

As input, the program takes in (as command line arguments), the employee.csv file, the time_cards.csv file, and the path to the expected output file pay stubs file. Every time the program is ran, it will update (overwrite) the employee.csv which keeps a history of total pay and taxes paid, and will generate a new pay stubs file. The final design of this program is up to you, given the specifications provided! As such, this is both an exercise in design and in implementation.

Provided Files

We have provided a few files for you. Please read this list carefully, as it will help you understand the requirements of the program.

Java Files

- **IEmployee.java** - an interface that all employees must implement. **DO NOT CHANGE THIS FILE**. We are providing this file as a contract for you to follow, and use it directly in our grading tests.
  - Within this file, each methods purpose and constraints are detailed in the javadoc. Make sure to read this carefully.
- **IPayStub.java** - an interface that all pay stubs must implement. You can change this file, as we only grade the final output of paystubs, but we needed to include the interface for IEmployee.
- **FileUtil.java**- This file contains methods that read and write lists from and to files. You can use this as is or you can modify it. Most students will leave this file unchanged. We provided this so you didn't have to know how to write to files in Java (at this time), but we will ask questions about it.
- **PayrollGenerator.java** - This is the main driver class for the program. You will need to implement the main method in this class. We have provided some guidance in the file on using FileUtil.java to read and write files. You do not have to use it this way.

    - DO NOT CHANGE the Arguments inner class, and make sure to use it in your code to get the file names. This will allow us to change the program arguments to test your code.
- **Builder.java** - Contains two methods you will want to implement. While you can add more, you shouldn't need to. Remember to add error checking to the methods as you are reading in file data.
- **ITimeCard** - You can update this file. Contains two suggested methods for TimeCard object.

Resources

We have provided a number of .csv files in the resources folder. These files are used to test your program. They are also the 'default' locations for files in your program. We suggest looking at them so you can come up with your own test files, especially time_cards.csv. It is a pretty happy path in what is in there, and you may want to write another time_cards.csv that has negative values and other edge cases.

Required Files and Constructors

In addition to the files provided, at a bare minimum (you will have more), you will need to implement the following classes:

- HourlyEmployee
- SalaryEmployee

The HourlyEmployee needs the following constructor

- `public HourlyEmployee(String name, String id, double payRate, double ytdEarnings, double ytdTaxesPaid, double pretaxDeductions)`

The SalaryEmployee needs the following constructor

- `public SalaryEmployee(String name, String id, double payRate, double ytdEarnings, double ytdTaxesPaid, double pretaxDeductions)`

> *[!TIP] The constructor is the same for both, but the classes are different.*

> *[!CAUTION] Don't forget that Employee.csv is in the order of*
>
> ```
> employee_type,name,ID,payRate,pretaxDeductions,YTDEarnings,YTDTaxesPaid
> ```
>
> *So be careful and think about what you are doing when you are reading in the contents of a line.*

:fire: Task 1: Design

Before you start writing, it is important to think about design. You DO NOT have to be perfect in your design, so we will come back to this step a few times.

1. First, become a detective and read through the files provided - both the .csv file and the java files. Take notes on what you are seeing (such as ordering of the csv lines). This is a common skill in software engineering, and you will need to do this often as you work with other people's code.
2. Go to DesignDocument.md and fill out (ONLY) the initial design section.

> *[!TIP] You are free to use mermaid or any other UML tools you want, just make sure if you are using another UML tool, you include the image in your submission. For a reminder on how to use mermaid markdown, look here.*

:fire: Task 2: Implement by Test Driven Development

After your initial design, you should seek to follow the TDD process. This means you should write tests first, and then implement the code to pass those tests. Or better stated, you should write *ONE* test first, implement, and repeat until you have written all your tests.

1. Figure out a number of tests by brainstorming (done in design)
2. Write **one** test
3. Write **just enough** code to make that test pass
4. Refactor/update as you go along
5. Repeat steps 2-4 until you have all the tests passing/fully built program

Note: you often don't know all the tests as you write. As such, it is alright to continue to expand your list. This is where people get stuck on TDD. They think they have to know **all** the tests before they start. You don't. You just need to know the next test, and then at the end you double check you have covered all code paths and have full coverage.

:raising_hand: Implementation Tips

Here are a few tips to help you implement the program:

1. An PayStub can have a reference to the employee (as compared to copying all the info).

2. You can use the `String.split(",")` method to split a line of a csv file into an array of strings. For example:

   ```java
   String line = "HOURLY,John Doe,12345,15.00,100.0,1000.00,100.00";
   String[] parts = line.split(",");`
   // then each part can be accessed by the index
   if(parts[0].equals("HOURLY")) {
       // do something
   }
   String name = parts[1]; // etc
   ```

   You can check to see if a line is correct by checking the length of the array.

   ```java
   if(parts.length != 7) {
       // throw an exception or something
   }
   ```

3. Don't forget to use try/catch around converting strings to doubles. If you don't, your program will crash if the file is not formatted correctly.

   ```java
   double payRate;
   try {
       payRate = Double.parseDouble(parts[3]);
   } catch(NumberFormatException e) {
       // handle the error
   }
   ```

4. You can use format, round, or BigDecimal to round to two decimal places.

```java
BigDecimal bd = new BigDecimal(value).setScale(2, RoundingMode.HALF_UP);
double rounded = bd.doubleValue();
```

   You may even just want to store values as BigDecimal, and convert to double on request. To do this, you need to use some of methods built into BigDecimal like `add`, `subtract`, `multiply`, and `divide`.

```java
public void doSomething(double val) {
   // assume value is a BigDecimal
   BigDecimal bd2 = new BigDecimal(val);
   this.value = value.add(bd2).setScale(2, RoundingMode.HALF_UP);
}

public double getValue() {
   return value.doubleValue();
}
```

5. If you want to add a method to make sure your code compiles, but not implement it yet, you can use `throw new UnsupportedOperationException("Not implemented yet")`. This will throw an exception if the method is called, but allow you to compile and test other parts of your code.

```java
public void doSomething() {
    throw new UnsupportedOperationException("doSomething() Not implemented
yet");
}
```

6. KEEP IT SIMPLE. The biggest down fall of all these assignments is that people try to over complicate it. Write each 'container' object. Make sure it works via tests, and then expand the features. The **last** thing you should update is the `main` method. However, everything else should be tested and working before that.

Above all, make sure you test each method as you write it. This is the key to TDD, but also will make your final testing much, much easier. Lastly, don't forget the design you learned in CS 5001! This isn't 100% new even if the syntax/language makes it feel all new. So it may help to ask yourself how you would do it in python.

Final Run Tests

We have provided a sample test that helps test the final implementation of your program. It is only one run, and not the only path (and only checks one file!). Use that as a template to write your own tests.

> *[!NOTE] A common question is "how many tests". The answer is that it isn't a number but that you test all paths, not just the "happy path". This means you need to test edge cases, and make sure everything is tested. Being complete in your testing, will make it much easier when working with the autograder (often common issues are style more than functionality if you are complete in testing).*
>
> *Also to be clear, you do not need to test the provided code - UNLESS you modify it! If you modify it, then you need to make sure there are tests for it.*

:fire: Task 3: Finish Design Document

By this point, your design has probably changed (very few people have perfect designs the first iteration). Update your design document with the final design in the "final design" section. We want to see the history of your first design to your final design. That is a good thing.

:fire: Task 4: Finish Report.md

Inside of Report.md you will need to answer a series of questions about your program, and about the learning objectives for the module in general. Fill it out.

> *[!IMPORTANT] A primary purpose of this activity is to get you working through a process in addition to writing code. In software engineering the process you follow is often just as important as the code you write. This is because the process is what allows you to work with others, and to be able to maintain and update code over time. It may seem tedious right now, but it is a skill that will pay off in the long run.*

Submission

When you are completed, you need to submit your code to gradescope. Go back to Canvas, and click through the link that takes you to the Gradescope assignment. From there you should be able to upload a .zip file of your project. Be mindful of how your .zip file is organized, it needs to be laid out as a functional project at the top level directory (as it should essentially be now).

Grading Rubric

1. Learning (AG)
   - Code compiles without issue
   - Code passes all tests
2. Approaching (AG)
   - Passes the style check.
3. Meets (MG)
   - README.md is filled out (name etc)
   - DesignDocument (INITIAL) sections are filled out
   - Proper use of polymorphic declaration for example: `List<IEmployee> employees = new ArrayList<>();`
   - All methods are tested with JUnit tests
   - Method contain proper javadoc comments (not just javadoc notation but proper wording in the comment)
   - Report.md technical questions are questions answered correctly.
4. Exceeds (MG)
   - Code is DRY (Don't Repeat Yourself)
   - Student uses proper inheritance without duplication for Employee Structures
   - Student uses proper inheritance without duplication for PayStub Structures
   - Methods include tests for edge cases in addition to happy path
   - Design document (FINAL) sections are filled out
     - The notation needs to be correct, and the TAs will double check the final design matches the final implementation.
   - Report.md Deeper Thinking question filled out

        - Includes at least two references/citations

Legend:

- AG - Auto-graded
- MG - Manually graded

Submission Reminder

For manually graded elements, we only guarantee time to submit for a regrade IF you submit by the DUE DATE. Submitting late may mean it isn't possible for the MG to be graded before the AVAILABLE UNTIL DATE, removing any windows for you to resubmit in time. While it will be graded, it is always best to submit by the due date, so you have full opportunity to improve your grade.

If you need a reminder about the grading policy, please review the syllabus and the canvas page on 'formative/summative' grading. This class uses a unique grading system that will allow you to be flexible with due dates and multiple resubmissions (if you submit with time for TAs to give feedback), but we also ask that you continue to work on the assignment until you get a full grade.

> *[!CAUTION] For this class, we give about a month for the **available until date**. This means you will only have a few manual resubmission attempts, and most everyone uses at least one! As such it is essential you submit on time!*

Autograder Limitation

Currently the autograder is limited in how it can test. As such, when it comes across an error it just stops. This means that if you have multiple errors in your code, you may only see the first one. We are working on improving this, but for now, you will need to fix the first error, and then rerun the tests to see the next error. Eventually, if every test passes, you will get the single point. It also may give you points for valid style, while errors exist in the code - so really assume the first 2 points are done together.

String Splitting

- String Split - W3Schools
- String Split Examples - Geeks4Geeks
- Java String Javadoc
- String Split Javadoc

Rouding to 2 Decimal Places

- Big Decimal Example
- Big Decimal Javadoc
- Number Formatting in Java

Reading Files

- Reading Files in Java
- Files Javadoc

Streams

- Streams Tutorial (more advanced)
- Make sure you go back to Team Activity 04 for a refresher on Streams

Testing File Contents

- JUnit @TempDir Example (more advanced)

The more common way in JUnit to test files is to make use of the @TempDir, it can be easy to test the contents of a file by reading it in and comparing it to a string. Here is an example of how you might do that:

```java
    @TempDir
    static Path tempDir;

    @Test
    public void testFinalPayStub() throws IOException {
        // copy employees.csv into tempDir
        Path employees = tempDir.resolve("employees.csv");
        Files.copy(Paths.get("resources/employees.csv"), employees);

        // get the path of the paystubs.csv
        Path payStubs = tempDir.resolve("paystubs.csv");



        String[] args = {"-e", employees.toString(), "-t",
                "resources/time_cards.csv",
                "-o", payStubs.toString()};

        // run main method
        PayrollGenerator.main(args);



      try {
        String expectedPayStubs = Files

.readString(Paths.get("resources/original/pay_stubs_solution_to_original.csv"));

        String actualPayStubs = Files.readString(payStubs);

        assertEquals(expectedPayStubs, actualPayStubs);
      } catch (IOException e) {
        fail("Error reading files");
      }

        // you could also read lines and compared the lists


    }
```

Remember when you test files, it is good to have multiple input/output files to test against! This requires setting a series of resources to help you out, and often manually calculating the expected output.