Assignment 3

Refer to Canvas for assignment due dates for your section.

Objectives:

- Implement solutions conforming to the object-oriented design principles of:
  - Encapsulation
  - Inheritance
  - Information hiding
  - Abstraction
- Write code that is modular and easy to extend
- Use inheritance to minimize code duplication
- Continue to meet the objectives of previous assignments

General Requirements

Create a new Gradle project for this assignment in your course GitHub repo. Make sure to follow the instructions provided in “Using Gradle with IntelliJ” on Canvas.

Create a separate package for each problem in the assignment. Create all your files in the appropriate package.

To submit your work, push it to GitHub and create a release. Refer to the instructions on Canvas.

Your repository should contain:

- One .java file per Java class.
- One .java file per Java test class.
- One build.gradle file for the whole project.
- One pdf or image file for each UML Class Diagram that you create. UML diagrams can be generated using IntelliJ or hand-drawn.
- All non-test classes and non-test methods must have valid Javadoc.

Your repository should **not** contain:

- Any .class files.
- Any .html files.
- Any IntelliJ specific files.

Suggestions for approaching this assignment

You only have one problem to work on this week because this assignment requires you to think very carefully about the design of your solution. An object-oriented solution will make ***extensive***

use of inheritance and have little to no code duplication between classes—this is what we will look for when grading your work.

The specification is long and complex. Read the **whole** problem then take time to plan your design before you start coding. Start by identifying commonalities across the various employees (see below) the system needs to account for. For example, look for those employees that could share fields or calculations. Only once you have a good sense of the relationships between the various employees should you start coding. If you find yourself writing the same code in multiple places, consider how you can refactor your design to make better use of inheritance and other OOD principles.

Problem 1

You are a part of a company developing automated tools for tracking, predicting and optimizing employees' productivity. **Employees' productivity** is a function of many complex parameters, but at this stage, your company models and estimates productivity as follows. The productivity tracking system distinguishes between two kinds of **employees**:

- Full-time employees
- Part-time employees

A full-time employee can be:

- Individual contributor
- Manager

A part-time employee can be:

- Benefits-eligible employee
- Hourly employee

For every **employee**, the productivity tracking system keeps track of:

- **Employee ID**, a unique employee identifier, represented as a `String`.
- **Contact info**, represented as a `ContactInfo`, a custom class that you will have to develop. Class `ContactInfo` contains the following fields.
    - **Name**, employee's first and last name, represented as a class `Name`, that you will also have to develop.
    - **Address**, an employee's address, represented as a `String`,
    - **Phone number**, the employee's phone number, represented as a `String`,
    - **Email address**, the employee's email address, represented as a `String`,
    - **Emergency contact**, the employee's emergency contact, represented as a `Name`.
- **Employment date**, the date the employee started working at their current company, represented as built-in class `LocalDate`.

- **Education level**, represented as an EducationLevel, a custom **enum** that you will have to develop, with possible values: **High school diploma, Some college, Associate's degree, Bachelor's degree, Master's degree, Doctoral or professional degree**.
- **Employment level,** represented as an `EmploymentLevel`, a custom **enum** that you will have to develop, with possible values: **Entry level, Intermediate level, Mid-level, Senior level, Executive level**.
- **Last year's earnings**, represented as a `Double`.

Additionally, for every **full-time employee**, the system keeps track of:

- **Base pay**, the employee's base pay, as defined by the contract, represented as a `Double`.
- **Bonuses**, the employee's last year earned bonuses, represented as a `Double`.
- **Overtime**, the employee's last year's earnings due to overtime payments, represented as a `Double`.
- **Date of the last promotion**, the date of the employee's last promotion, represented as a `LocalDate`.
- **Number of projects**, the number of projects an employee is working on, represented as an `Integer`.

Further, for every **part-time employee**, the system keeps track of:

- **The contractual number of worked hours**, denoting the number of hours an employee is contractually expected to work per week, represented as a `Double`.
- **The actual number of worked hours**, denoting the actual number of hours an employee worked last week, represented as a `Double`.
- **Bonus and overtime earnings**, denoting the combined bonus and overtime earnings, represented as a `Double`.

Further, for every **manager**, the system keeps track of **the number of employees they manage**, represented as an `Integer`.
Additionally, for every **individual contributor**, the system keeps track of:

- **The number of patents**, denoting the number of patents awarded to the employee while with the company, represented as an `Integer`.
- **The number of publications**, denoting the number of publications that the employee published /presented last year, represented as an `Integer`.
- **The number of external collaborations**, denoting the number of external projects that the employee is involved in, represented as an `Integer`.

Finally, for every **hourly employee**, the system keeps track of:

- **Hourly earnings**, denoting the contractual hourly earnings, and represented as `Double`.

The system estimates an employee's productivity according to the following rules:

- **Base productivity estimate for full-time employees:** Base productivity of full-time employees is calculated as a ratio between an employee's last year's earnings, and their base pay.
- **Base productivity estimate for part-time employees:** Base productivity of part-time employees is calculated as a ratio between an employee's actual number of worked hours, and their contractual number of worked hours, and the result is multiplied by 3.7.
- **Number of projects bonus:** Every full-time employee involved in more than 2 projects gets a bonus boost where 1.5 is added to their base productivity estimate.
- **Manager bonus:** Every manager who manages more than 8 employees gets a bonus boost, where 1.8 is added to their current productivity estimate.
- **Individual contributor bonus:** Every individual contributor who published more than 4 publications last year, gets a bonus boost, where 1.3 is added to their current productivity estimate.
- **Employment level bonus:** If an employee is hired into the intermediate level role, they get a bonus boost, where 1.4 is added to their current productivity estimate.
- **Last promotion penalty:** For every full-time employee, if more than three years have passed since their last promotion, they get a penalty, where 0.8 is subtracted from their current productivity estimate.
- **Hourly earnings bonus:** If an hourly employee's hourly rate is less than $14, they get a bonus boost, where 3 is added to their current productivity estimate.

Design and implement method `estimateProductivity`(), whose return type is `Double`. That means that when we call method `estimateProductivity`() on some `Employee`, it should return an estimated productivity for that employee as `Double`.

Don't forget to write tests and generate a UML diagram!