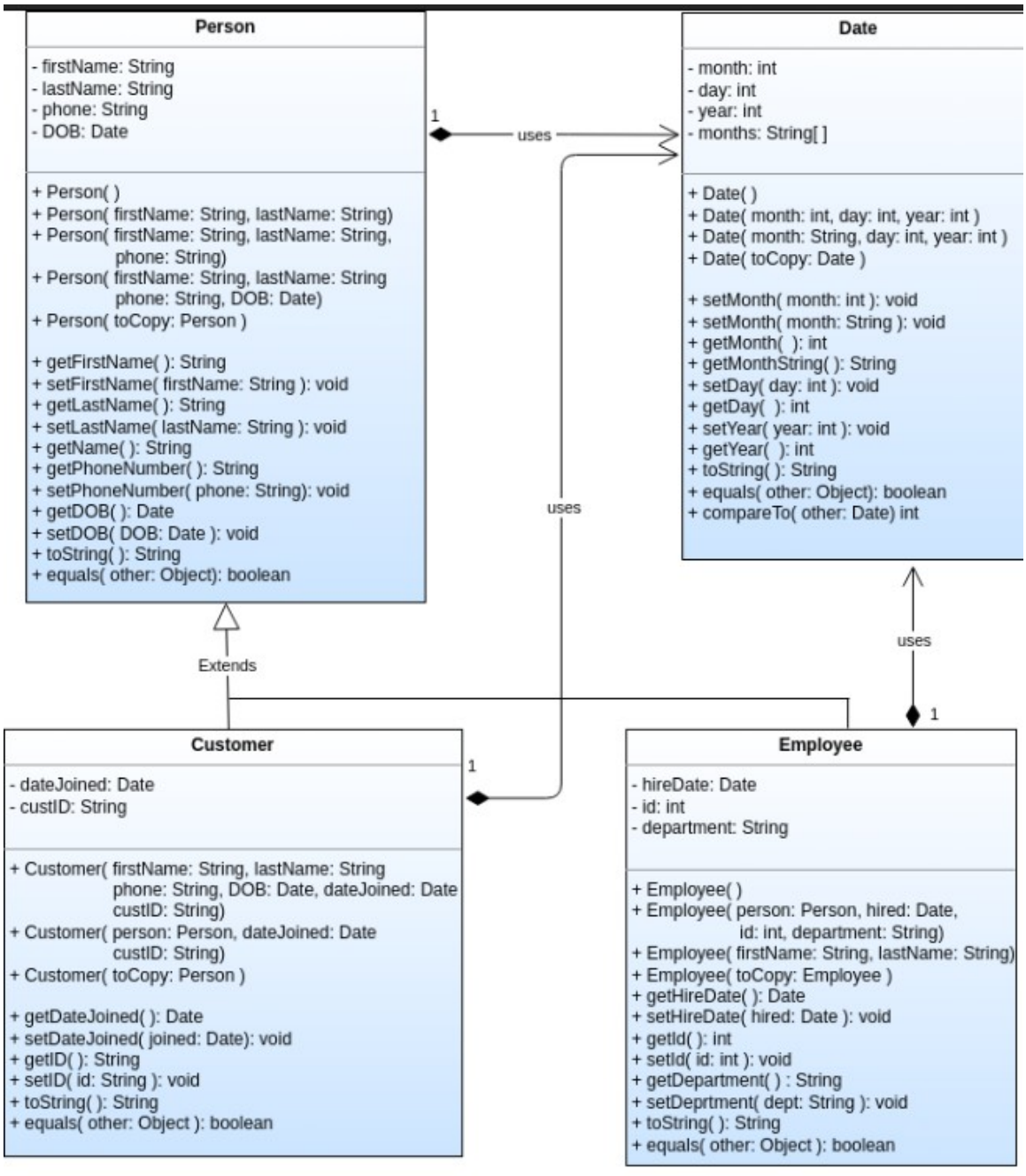
CSCI165 Computer Science II
Lab Assignment: OOP Inheritance, Composition and polymorphism

Objectives

- Work with inheritance and composition
- Override methods with the **@Override** annotation
- Identify and fix **NullPointer** situations in an inheritance hierarchy
- Design simple polymorphic solution to a problem
- Pay close attention to detail



You have been provided with the following

- Class Person
- Class Employee as a subclass of Person
- Class Date
- Class Account
- A ***drawio*** file that contains the UML specification for these classes, showing composition and inheritance.

Task One: The Employee class

- This class illustrates using both composition and inheritance to define a new class.
    - **Inheritance**: Employee ***is a*** Person, so Employee is a subclass of Person
    - **Composition:**
        - Person ***has a*** date of birth, so Person has an instance feature of type Date
        - Employee ***has a*** date of hire, so Employee has an instance feature of type Date

- Experiment with the Employee class by creating a driver class and creating some instances of Employee. ***Demonstrate that you can use all of the constructors.*** Print some information about your Employee instances by calling toString.

  Do you run into NullPointer situations here? Write some comments in your Driver file describing these issues if they exist.

  If these issues are present, what do you think would be a good solution? **DO THIS** and then provide comments on how you solved the problem and demonstrate that the problem is solved.

  Note: There are multiple approaches to solving these issues, so let's talk about them.

Task Two: The Customer class

- Create the class Customer as a ***subclass*** of Person (or re-use and modify the Customer class from earlier labs). This class illustrates using both composition and inheritance to define a new class.
    - **Inheritance**: Customer ***is a*** Person, so Customer is defined as a subclass of Person and can be defined without repeating "Person" stuff.
    - **Composition:**
        - Person ***has a*** date of birth
        - Customer ***has a*** date that tracks when they joined (created their account)

- Include the following
    - **Properties:**
        - dateJoined: Date
        - custID: String

- **Methods:**
  - Privacy protection must be in place for all mutable objects declared **private**
  - Getters and setters for dateJoined and custID.
  - Overridden toString and equals, both marked with the **@Override** annotation. Both should also call super when appropriate

- **Constructors:**
  - Overloaded to accept: firstName, lastName, Phone, DOB, dateJoined, custID
  - Overloaded to accept: Person, dateJoined, custID
  - Copy constructor
  - Any constructor you feel fits the need
  - All constructors should call **super( )** and **this( )** where appropriate to reuse constructors and avoid redundancy.

- **UML:** The UML class diagram for Customer has been included in the ***drawio*** file for you. If you notice any typos (probable, apologies) make a common sense decisions or post a question on Discord so everyone can benefit from the discussion.

- **Unit Tests:** A collection of JUnit tests has been provided for you. Your Customer class is required to get all passes for these tests. This is how I will be grading your Customer class, so be sure to run these tests against your code.

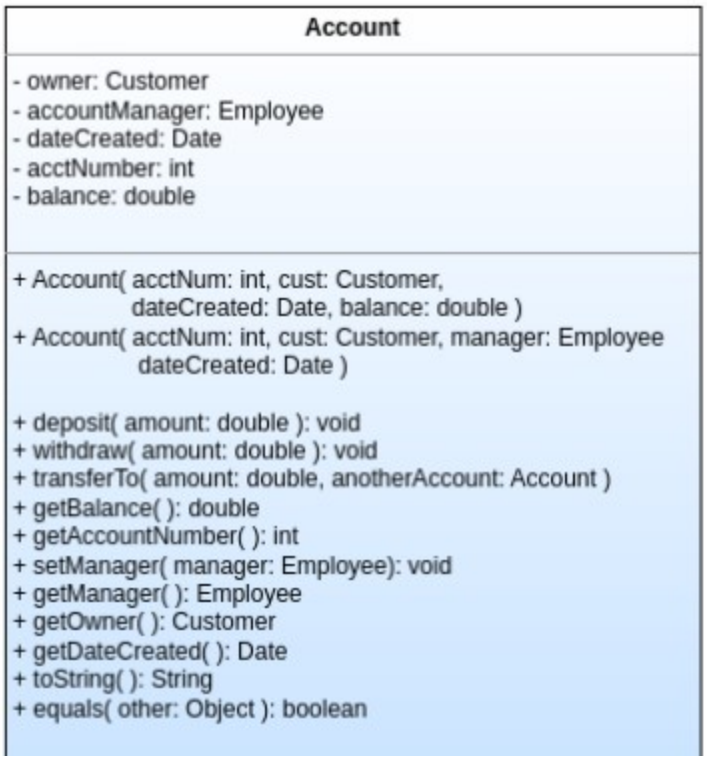
- **Driver:** Using the same Driver class from Task One, prove that you can create some instances of the Customer class. Print some information from the instances and show you can call inherited methods. Be thorough with your proof of concept.

Task Three: The provided Account class

Composition:

- An account ***has an*** Owner (Customer)
- An account ***has a*** manager (Employee)
- An account ***has a*** date of creation (Date). This date should always be equal to or after the **dateJoined** field in the Customer class

Using the same Driver class from above, and the provided ***Account.java*** class, experiment with creating some account instances. Demonstrate that you can



create account instances using all of the constructors and print out some information about the account.

UML: Expand the UML diagram to include the Account class into the overall architecture. Use the correct composition and inheritance notation.

Note: You do not have to show String object composition or any other Java API objects unless they are part of a class' instance features. If you use a Scanner object in a method yo do not need to show this on the diagram.

Using the ***Account.java*** class as a base class, write two derived classes called

- ***SavingsAccount***
- ***CheckingAccount***

A ***SavingsAccount*** object, in addition to the attributes inherited from the Account class, should have an interest percentage variable and a method (**addInterest**) which adds interest to the account. Don't worry about implementing any fancy finance rules or mathematics here. The object oriented structure is what is of interest. Obviously you can add these algorithms if you'd like.

A ***CheckingAccount*** object, in addition to the attributes of an Account object, should have an overdraft limit variable and a method (**isInOverDraft**) that returns true if the account is in overdraft. You may need to add additional variables to manage this. **Make design choices of your own and please add documentation explaining your choices.**

> **Definition:** An **overdraft** is an extension of credit from a lending institution that is granted when an account reaches zero. The bank we are writing this software for allows customer to withdraw from checking past a balance of zero up to a certain amount. This is what the "overdraft limit" represents. Override the withdraw method in ***CheckingAccount*** to include this behavior.

For both Checking and Savings Account classes

- Privacy must be protected for any mutable object that has been marked with ***private***
- Add an appropriate collection of constructors (including a copy constructor) to allow the creation of the accounts. Call ***super()*** and ***this()*** when appropriate to reuse constructors.
- You should add a copy constructor to the Account class to facilitate hierarchy copying
- Appropriately override ***toString*** to call **super's** toString and add the appropriate sub class data.
- Appropriately override ***equals*** to call **super's** equals method and add the appropriate sub class tests. Both of these methods must have the ***@Override*** annotation
- Remember that toString and equals should travel up the inheritance hierarchy dealing with all fields. If each level above has properly defined versions, this is a trivial process.

UML: Add class diagrams for ***CheckingAccount and SavingsAccount*** to the ***drawio*** file. Connect the new class diagrams (and Account) to the existing classes and add the appropriate composition and inheritance notation to create a complete system diagram. You can increase the page size in the application to contain more objects, or you can create a series of related pages using the shorthand notation and the full class notation.

Unit Tests: Add a file to test both the Checking and Savings account classes. I am particularly interested in seeing robust tests for the copy constructors and equals methods as these are the most structurally complex.

Task Four: The Bank Class

Now create a Bank class. This class will contain an array (or ArrayList) of Account objects. Accounts in the array could be instances of the ***Account*** class, the ***SavingsAccount*** class, or the ***CheckingAccount*** class. This organization is the foundation for achieving polymorphic behavior.

- Write an **update** method in the bank class. It will iterate through each account, and update in the following ways:
    - **Savings Accounts** get interest added (via the method you already wrote);
    - **Checking Accounts** get a letter sent (message printed) if they are in overdraft.
    - You have two options here
        1. Perform type checks to determine which method to call based on the origin (clunky, inefficient and not adaptive to future enhancements). **Not best practice!**
        2. Design a polymorphic solution that alleviates the need for type checking. **Best practice!**
    - The polymorphic solution is the desired approach because it removes the need for type checking and is "future proof" because you will be able to extend the inheritance hierarchy and add new classes in the future and the code in the Bank class will not have to be updated. The goal here is to understand that a polymorphic solution in our Account hierarchy will allow the Bank class to adapt to **any new classes** without modification.

      Even with Account subclasses that have not been written yet.
    - Your task here is to design a polymorphic solution to this problem. It will require you to modify the design of your classes a bit. I want to see the Bank's **update** method be able to iterate through a collection of Accounts and update each account accordingly **without any type checks.**
    - I also want you to describe your design choices in detail, somewhere obvious in the comments. Include sufficient comments to convince me that you understand this polymorphic approach
- The Bank class also requires methods for opening and closing accounts, and for paying a dividend into each account. The dividend can be handled by making a deposit into each account based on a percentage. (You can make these values up as the actual amounts are not important to this exercise)

You design this and include a description of your design choices as comments.

- **Prove that these work:** Create a separate Driver file, create some test accounts (some of each type) and print some information about each one. Include sufficient printed messages to communicate what is happening.

Submission: Push all files mentioned in this lab to your repo.