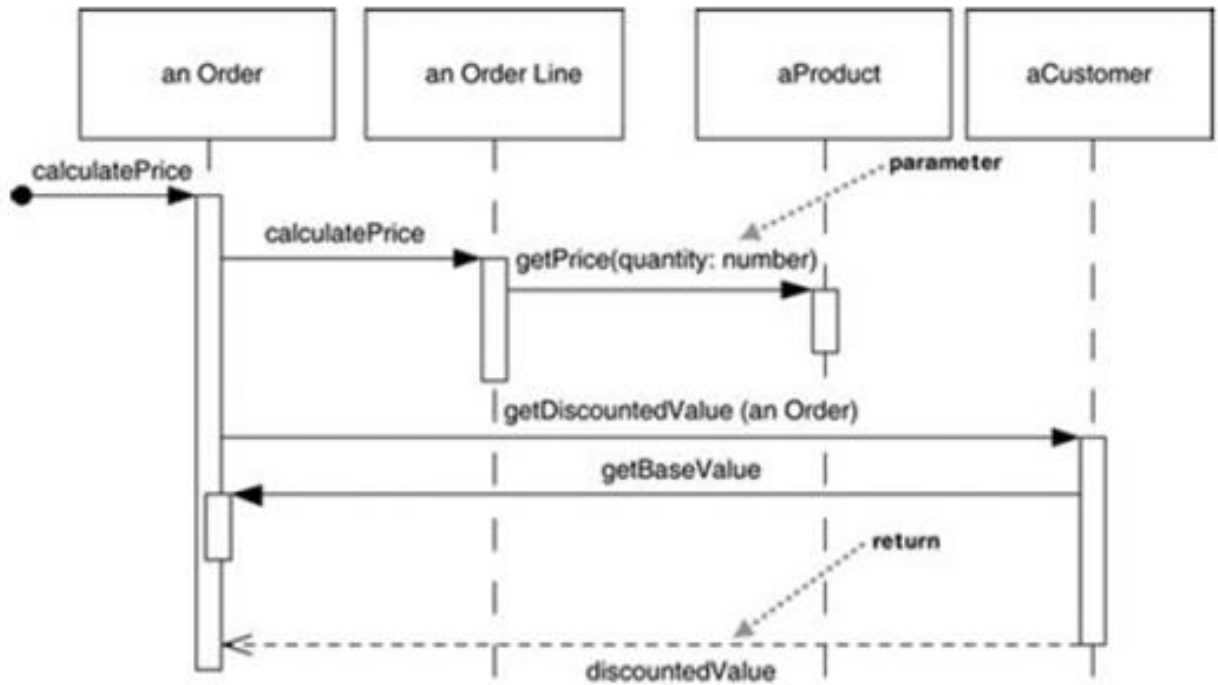# Lab 4 Parts A, B

A. Re-do the sequence diagram on Slide 16 of Lesson 4 (reproduced below).

In your diagram, be sure to include the following:

- Message numbering
- Proper UML syntax for the objects displayed at the top.
- Iteration markers where looping occurs

B. Create a sequence diagram based on the flow that occurs when an actor invokes the checkoutBook() method on CheckoutForm

```
//FROM CLASS CheckoutForm
   public void checkoutBook()   {

       theCheckoutController.checkoutBook(m_book, m_member);
       displayCheckoutInfo();
       clearCheckoutFields();
   }
```

```
//FROM CLASS CheckoutController
   public void checkoutBook(Book book, LibraryMember member) {

       CheckoutRecord aCheckoutRecord = new CheckoutRecord();
       aCheckoutRecord.setDueDate(member.getCheckoutPeriod());
       aCheckoutRecord.addBook( book );
       member.addCheckoutRecord( aCheckoutRecord );
       theILibraryDBSubsystem.addCheckoutRecord(member.getMemberID(),
                                    aCheckoutRecord);


   }
```

Lab 4 C
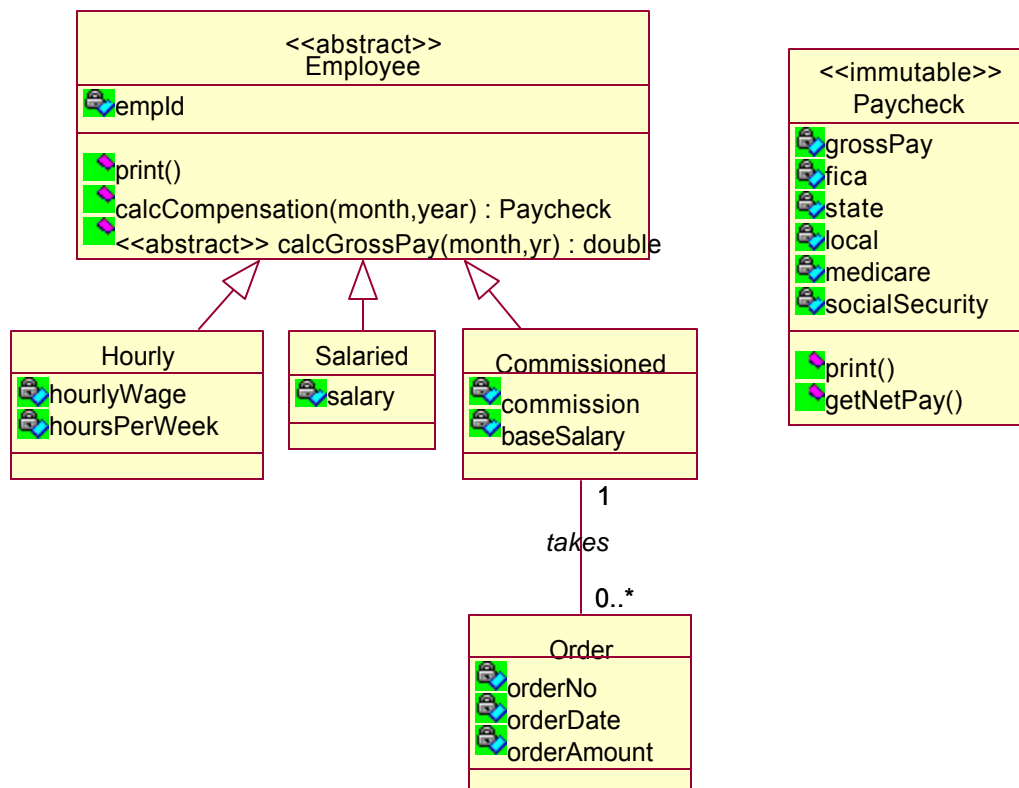# POLYMORPHISM
Payroll Calculation

Objectives:
- Practice implementing and using polymorphic methods.
- Understand how the Template Method design pattern works.

Human Resource Application:

The HR department has identified three types of Employees based on how they are paid: Hourly, Salaried, and Commissioned. The paycheck for each type is calculated differently:

- Hourly employees are paid monthly and their paycheck amount is calculated based on their hourly wage and the hours per week they work. For simplicity just assume four weeks for each month.

- Salaried employees are paid monthly and their paycheck amount is a fixed amount every month.

- Commissioned employees are also paid monthly. They receive a small base salary, plus a percentage (commission) on the total value of all orders they sold during the previous month.

Lab 4
Page 2
POLYMORPHISM
Payroll Calculation

Tasks:

1. Add a concrete `calcCompensation()` method to Employee.

   a. This method takes the month and year as arguments for which to calculate the compensation.
   b. For hourly and salaried employees the amount is the same every mont h.
   c. For Commissioned employees the amount depends on the orders they sold that month.

2. The `Employee.calcCompensation()` method delegates to the respective derived class to calculate the gross pay amount by invoking the abstract `Employee.calcGrossPay()` method.

3. The `Employee.calcCompensation()` method then calculates the FICA, state & local taxes, medicare and social security contributions based on the gross pay. Assume the following fixed tax percentages:

   - FICA is 23%
   - State tax is 5%
   - Local tax is 1%
   - Medicare is 3%
   - Social Security is 7. 5%

Important details about the UML class diagram:

- `Paycheck` class are immutable, i.e. all data needs to be pas sed to the constructor and no setter methods should be provided.

- `Employee` is an abstract class!

- `Employee.calcCompensation()` returns a `Paycheck` object!

- In order to calculate the payche ck for a Commissioned employee you need to access all the `Order` objects that each Commissioned employee is responsible for and add up the order amount of all orders during a given month.

Lab 4, Part D

D. Create a sequence diagram for the problem described in Lab 4, Part C. Create a distributed control solution. As you distribute control, make sure that the object that handles a step of processing really should be responsible for that behavior, based on the purpose of the class that was determined in the class diagram.

Lab 4, Part E
[March 2017 Programming Exam Question]

In a company, employees may have multiple bank accounts: zero or more savings accounts and zero or more checking accounts. Each checking account has an account id, a balance, and a monthly fee. Each savings account has an account id, a balance, and an interest rate associated with the particular type of savings account. It is possible to read the current balance in any of these accounts, but it is also possible to determine the balance after interest or monthly fee is applied by calling the `computeUpdatedBalance` method on the account.

An administrator has access to all employee records and from time to time computes the total balance across all employee-owned accounts; for each account, the balance that is needed in this computation is the *updated balance*. This computation is performed in the static method

$$\text{computeUpdatedBalanceSum}$$

in the `Admin` class.

Below is a class diagram showing the classes involved and relationships between them. A sequence diagram for the operation `computeUpdatedBalanceSum` is also provided. Your task in this problem is to write Java code that implements the classes and relationships shown in the diagram. Shells for the `Admin` and the `Employee` classes have been provided in your workspace. Also, a `Main` class (with a `main` method) has been provided for you to test your code (in the `launch` package) – the code in the `main` method has been commented out; when you are ready to test your code, you can uncomment it.

The method `computeUpdatedBalance` in `CheckingAccount` does the following computation to obtain the return value:
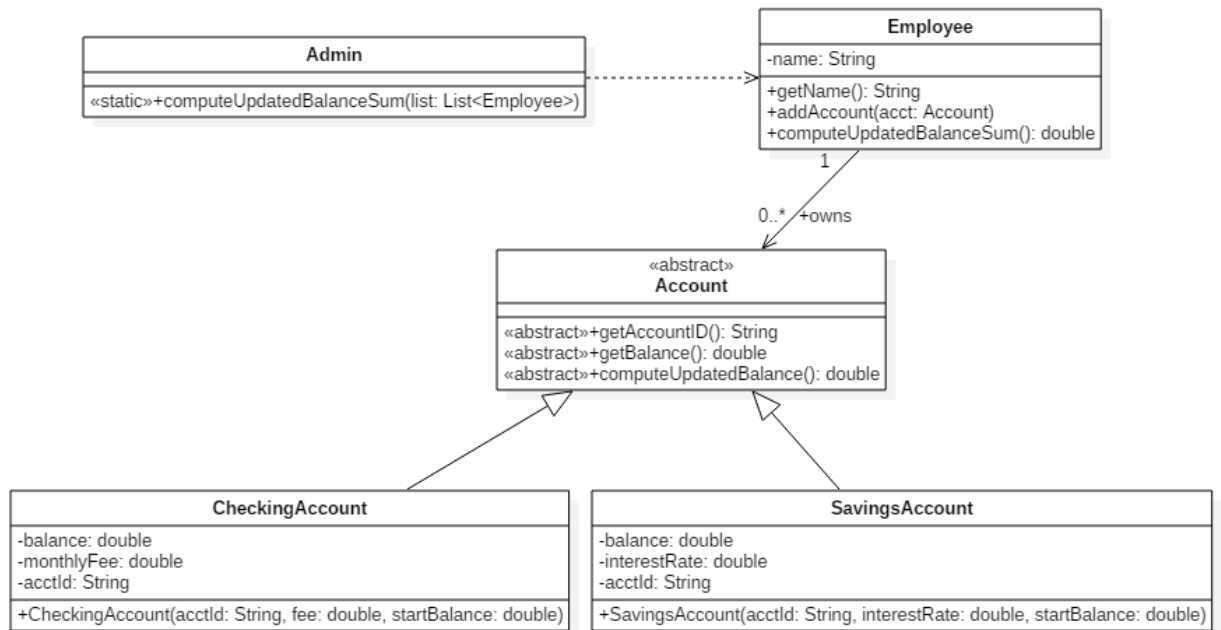
$$\text{balance} - \text{monthlyFee}.$$

The method `computeUpdatedBalance` in `SavingsAccount` performs the following computation to obtain the return value:

$$\text{balance} + (\text{interestRate} * \text{balance}).$$

Points to notice about the class diagram:

1. `CheckingAccount` and `SavingsAccount` are subclasses of `Account`. Also, `Account` has several abstract methods which must be implemented in its subclasses.
2. There is a one-way association from `Employee` to `Account`. It is important that your code reflects and maintains this association.
3. The diagram has a mix of dependencies and associations; make sure your code distinguishes between these properly.

You may not modify the signatures or qualifiers of the methods contained in the `Admin` or `Employee` classes that have been provided. You will need to create all the other classes mentioned in the diagrams.

.

Admin

«static»+computeUpdatedBalanceSum(list: List<Employee>)

Employee

-name: String

+getName(): String
+addAccount(acct: Account)
+computeUpdatedBalanceSum(): double

1

0..* +owns

«abstract»
Account

«abstract»+getAccountID(): String
«abstract»+getBalance(): double
«abstract»+computeUpdatedBalance(): double

CheckingAccount

-balance: double
-monthlyFee: double
-acctId: String

+CheckingAccount(acctId: String, fee: double, startBalance: double)

SavingsAccount

-balance: double
-interestRate: double
-acctId: String

+SavingsAccount(acctId: String, interestRate: double, startBalance: double)

interaction Sequence Diagram for computeUpdatedBalanceSum

Admin

Employee

Account

1 : computeUpdatedBalanceSum(list:List<Employee>)

2 : * computeUpdatedBalanceSum

3 : * computeUpdatedBalance