Inheritance - Pairs Exercise

The purpose of this exercise is to provide you the opportunity to practice writing code that makes use of the Object-Oriented Programming principle of inheritance.

This exercise is comprised of three parts. Parts 1 and 2 are to be completed on the first day and are associated with the first day of the inheritance lecture. Part 3 is to be completed on the second day after the second lecture on inheritance.

Learning Objectives

After completing this exercise, students will understand:

- How to create "super" classes.
- How to "inherit" behavior and attributes in child classes.
- How to appropriately use abstract classes.

Evaluation Criteria & Functional Requirements

- The project must not have any build errors.
- Code is presented in a clean, organized format.
- Code is appropriately encapsulated.
- Inheritance is being used appropriately to avoid code duplication.
- The code meets the specifications defined below.

Bank Teller Application

Notes for All Classes

- X in the set column indicates it **should have a `set` accessor**.
- Nothing in the set column indicates the attribute is derived.
- Readonly attributes do not require a `set` accessor.
- Private setters require the property to have `private set` .

Part 1 (Day 1)

Create three new classes to represent a bank account, savings account, and a simple checking account.

BankAccount

The BankAccount class represents a generic account at a bank. It contains the most basic information about an account.

Implement the `BankAccount` class.

Constructor	Description
BankAccount()	A default bank account's balance is set to a 0 dollar balance.
	A default bank account's accountNumber is defaulted to "unknown".
BankAccount(String acctNum, BigDecimal bal)	2-argumentt ctor; Set accountNumber and balance to values passed

Attribute Name	Data Type	Get	Set	Description
accountNumber	String	X	X	Returns the account number that the account belongs to.
balance	BigDecimal	X	private	Returns the balance value of the bank account in USD.

Method Name	Return Type	Description
deposit(BigDecimal amountToDeposit)	BigDecimal	Adds `amountToDeposit` to the current balance, and returns the new balance of the bank account.
withdraw(BigDecimal amountToWithdraw)	BigDecimal	Subtracts `amountToWithdraw` from the current balance, and returns the new balance of the bank account.
transfer(BankAccount destinationAccount, BigDecimal transferAmount)	void	Withdraws `transferAmount` from this account and deposits it into `destinationAccount` .

CheckingAccount

Implement the `CheckingAccount` class.

A `CheckingAccount` "is-a" `BankAccount` , but also has some additional rules:

Override Method	Description
withdraw	If the balance falls below $0.00 a $10.00 overdraft fee is also withdrawn from the account.
withdraw	Checking account cannot be more than $100.00 overdrawn. If a withdrawal is requested leaving the account more than $100.00 overdrawn, it fails and the balance remains the same.

SavingsAccount

Implement the `SavingsAccount` class.

A `SavingsAccount` "is-a" `BankAccount` , but also has some additional rules:

Attribute Name	Data Type	Get	Set	Description
interestRate	BigDecimal	X	private	static data member - set to 2%

Override Method	Description
withdraw	If the current balance is less than $150.00 when a withdrawal is made, an additional $2.00 service charge is withdrawn from the account.
withdraw	If a withdrawal is requested for more than the current balance, the withdrawal fails and balance remains the same. No fees are incurred.
addInterest	Add calculate interest (balance * interestRate) and add to the balance

Sample usage

```
BankAccount aBankAccount = new BankAccount();
BankAccount anotherAcct  = new BankAccount("031952",new BigDecimal("19.95"));

BankAccount aCheckingAccount = new CheckingAccount();
BankAccount anotherCheckAcct = new CheckingAccount("083112", new BigDecimal("14.92"));
BankAccount aSavingsAccount  = new SavingsAccount();
BankAccount anotherSaveAcct  = new SavingsAccount("07041776", new BigDecimal("17.89"));

BigDecimal amountToDeposit = new BigDecimal("1.00");
BigDecimal newBalance = aCheckingAccount.deposit(amountToDeposit);

BigDecimal amountToTransfer = new BigDecimal("0.50");
aCheckingAccount.transfer(aSavingsAccount, amountToTransfer);

aSavingsAccount.addInterest();
```

Getting Started

- Import the inheritance-exercises-pair project into Eclipse.
- A BankTeller.java applicatiom program has been provided for use in testing your classes
- Add the appropriate classes to to the project satisfy the requirements.

Tips and Tricks

- A good way to determine if you are implementing inheritance correctly is to read the code or classes out loud. A child class "is-a" type of its parent. For instance, a CheckingAccount "is-a" BankAccount. Thinking about the relationships of objects in these terms will help you to quickly identify opportunities to improve your code.
- We will be learning about a principle over the next few days called polymorphism. This isn't something you need to know to do the work in this exercise, but you may want to revisit this code after learning about it to see how this concept may have changed your overall design.
- We will also be learning about a concept called abstract classes. Again, after learning this concept, how might this change your approach to the solution you provide?

Part 2 (Day 1)

Create a bank customer that has bank accounts, as well as a command line application to verify your code is working as expected.

BankCustomer

Implement the `BankCustomer` class.

Attribute Name	Data Type	Get	Set	Description
name	String	X	X	Returns the account holder name that the account belongs to.
address	String	X	X	Returns the account number that the account belongs to.
phoneNumber	String	X	X	Returns the account number that the account belongs to.
accounts	BankAccount[]	X		Returns the customer's list of BankAccounts as an array.

Method Name	Return Type	Description
addAccount(BankAccount newAccount)	void	Adds newAccount to the customer's list of accounts.

Sample usage

```
BankAccount checkingAccount = new CheckingAccount();
BankAccount savingsAccount = new SavingsAccount();

BankCustomer jayGatsby = new BankCustomer();
jayGatsby.addAccount(checkingAccount);
jayGatsby.addAccount(savingsAccount);

System.out.println(String.format("Jay Gatsby has %s accounts.", jayGatsby.getAccounts().length)) // Jay Gat
```

BankTeller

Use the BankTeller application program class provided to verify that your application is working as expected.

Things you should verify:

- Objects of all classes are instantiated correctly
- The withdraw method works as expected for Checking and Savings accounts.
- The deposit method works as expected.

- The addInterest method works as expected for the Savings accounts.
- Accounts can be addded to a bank customer.
- Determining if a customer is a VIP works as expected. (Part 3)

Part 3 (Day 2)

Customers whose combined account balances are at least $25,000 are considered VIP customers and receive special privileges.

Add a method called `getIsVIP` to the `BankCustomer` class that returns true if the sum of all accounts belonging to the customer is at least $25,000 and false otherwise.

Getting Started

- Import the inheritance-exercises-pair project into Eclipse.
- Add the appropriate classes to satisfy the requirements.

Tips and Tricks

- When adding accounts to customers, there is a data structure we learned about that makes it a lot easier to add items. Keep in mind, if you have appropriately encapsulated your attributes, the type of object you use to store the accounts doesn't need to be the same as the type that is returned from the getAccounts method.
- We will be learning about a principle over the next few days called polymorphism. This isn't something you need to know to do the work in this exercise, but you may want to revisit this code after learning about it to see how this concept may have changed your overall design.
- We will also be learning about a concept called abstract classes. Again, after learning this concept, how might this change your approach to the solution you provide?
- A good way to determine if you are implementing inheritance correctly is to read the code or classes out loud. A child class "is-a" type of its parent. For instance, a CheckingAccount "is-a" BankAccount. Is a BankCustomer a BankAccount, or does a BankCustomer have a BankAccount? Thinking about the relationships of objects in these terms will help you to quickly identify opportunities to improve your code.