Polymorphism

The purpose of this exercise is to practice writing code that uses the Object-Oriented Programming principle of polymorphism.

Learning objectives

After completing this exercise, you'll be able to:

- Explain the concept of polymorphism and how it's useful
- Demonstrate an understanding of where inheritance can assist in writing polymorphic code
- State the purpose of interfaces and how they're used
- Use polymorphism through inheritance using IS-A relationships
- Use polymorphism through interfaces using CAN-DO relationships
- Give examples of interfaces from the Java/C# standard library (Collections)

Evaluation criteria and functional requirements

- The project must not have any build errors.
- Code is presented in a clean, organized format.
- Code is appropriately encapsulated.
- Polymorphism is used appropriately to avoid code duplication.
- The code meets the specifications defined in the remainder of this document.

Bank customer application

Notes for all classes and interfaces

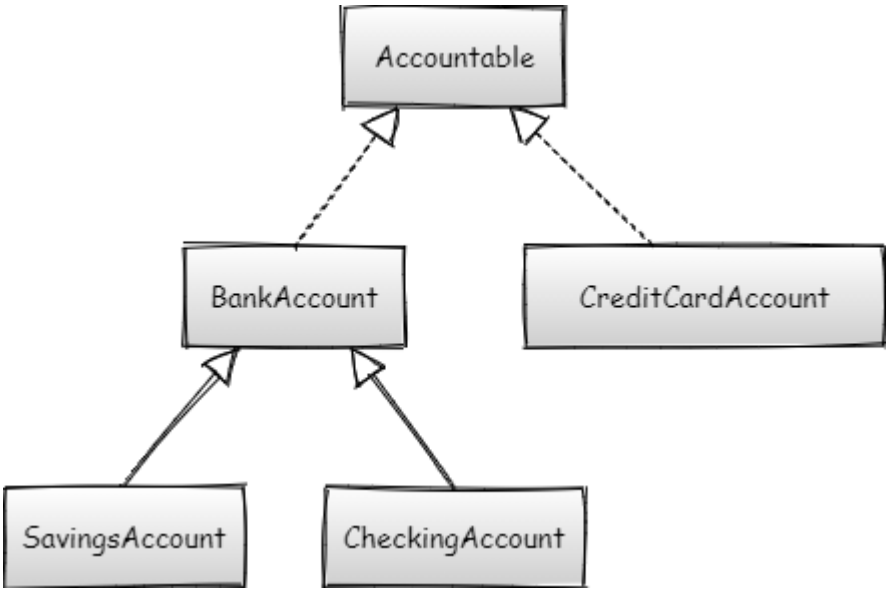
- X in the set column indicates it **must have a Setter**.
- Nothing in the set column indicates the attribute is derived.
- Readonly attributes don't require a Setter.

Instructions

This code is from the Inheritance day. The bank account classes work well, but now the bank needs to calculate a customer's total assets to assign them VIP status if they have $25,000 or more in assets at the bank.

The bank is also introducing credit cards. Since credit cards aren't strictly bank accounts that store money, they don't inherit from the `BankAccount` class. However, you must still account for them in the VIP calculation.

For this exercise, you'll add new features to the code to create a `BankCustomer` class that has multiple accounts. You'll also create a new type of account: a credit card account. A credit card account isn't a `BankAccount`, but you must store it with the customer as one of their accounts. To do this, you need to create a new interface that specifies that an object is `Accountable` and has a `getBalance()` method.



For this exercise, you will:

1. Add a new method to allow customers to transfer money between `BankAccount`s.
2. Create a new interface called `Accountable`.
3. Make `BankAccount` implement `Accountable`.
4. Create a new class called `CreditCardAccount` that's also `Accountable`.
5. Create a `BankCustomer` class that has many `Accountable` objects.
6. Add an `isVip()` method to `BankCustomer`.

Step One: Add a new `transferTo()` method to transfer money between `BankAccount`s

Add the following method to allow `BankAccount`s to transfer money to another `BankAccount`. Where would you add this method to make sure it works for all `BankAccount`s, including `SavingsAccount` and `CheckingAccount`?

Method Name	Return Type	Description
`transferTo(BankAccount destinationAccount, int transferAmount)`	`int`	Withdraws `transferAmount` from this account, deposits it into `destinationAccount`, and returns the new balance of this account.

New unit tests have been added for this section. This section is complete when the `CheckingAccountTest`, `SavingsAccountTest`, and `BankAccountTest` unit tests all pass.

> Note: Initially, the unit tests may show `transferTo()` as a ***missing method error*** rather than a failing test until you add the method. Once added, the `transferTo()` method may still fail. It won't be because it's missing.

Step Two: Create the `Accountable` interface

The `Accountable` interface means that an object can be used in the accounting process for the customer.

Method Name	Return Type	Description

Method Name	Return Type	Description
`getBalance()`	`int`	Returns the balance value of the account in dollars.

Step Three: Make the `BankAccount` class accountable

Add the `Accountable` interface to `BankAccount`, making `BankAccount` and all the classes that inherit from `BankAccount` accountable classes.

Step Four: Implement a new `CreditCardAccount` class

A `CreditCardAccount` isn't a `BankAccount` but "can-do" `Accountable`.

Constructor	Description
`CreditCardAccount(String accountHolder, String accountNumber)`	A new credit card account requires an account holder name and account number. The debt defaults to a 0 dollar balance.

Attribute Name	Data Type	Get	Set	Description
`accountHolder`	`String`	X		The account holder name that the account belongs to.
`accountNumber`	`String`	X		The account number that the account belongs to.
`debt`	`int`	X		The amount the customer owes.

Method Name	Return Type	Description
`pay(int amountToPay)`	`int`	Removes `amountToPay` from the amount owed and returns the new total amount owed.
`charge(int amountToCharge)`	`int`	Adds `amountToCharge` to the amount owed, and returns the new total amount owed.

> Note: Be sure to implement the interface. The balance for the accounting must be the debt as a negative number.

Once the `CreditCardAccountTest` unit tests pass, this step is complete.

Step Five: Implement the `BankCustomer` class

Implement the `BankCustomer` class. A bank customer has a list of `Accountable`s.

Attribute Name	Data Type	Get	Set	Description
`name`	`String`	X	X	The account holder name that the account belongs to.
`address`	`String`	X	X	The address of the customer.

Attribute Name	Data Type	Get	Set	Description
`phoneNumber`	`String`	X	X	The phone number of the customer.
`accounts`	`List<Accountable>`	X*		The customer's list of Accountables.

Method Name	Return Type	Description
`getAccounts()`	`Accountable[]`	Returns an array of the customer's accounts.
`addAccount(Accountable newAccount)`	`void`	Adds `newAccount` to the customer's list of accounts.

* Note: The `getAccounts()` method returns an array, but since you need to add accounts whenever the `addAccount()` method is called, you'll use a `List` to store the accounts.

Step Six: Add the `isVip()` method to the `BankCustomer` class

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

Add a method called `isVip` 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.

Once the `BankCustomerTest` unit test passes, this section is complete.