Inheritance exercise

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

Learning objectives

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

- Describe the purpose and use of inheritance in an Object-Oriented Programming environment
- Define and use superclasses and subclasses in an inheritance hierarchy
- Identify superclasses and subclasses from viewing source code
- Define an IS-A relationship in reference to inheritance
- Define what overriding means in the context of inheritance
- Describe what's being inherited
- Describe how access modifiers work in an inheritance relationship
- Use super class constructors in a subclass

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.
- Inheritance is used appropriately to avoid code duplication.
- The code meets the specifications defined in the remainder of this document.

Bank teller application

Notes for All Classes

- All attributes have `private` access.
- X in the get column indicates the attribute *must have a `get` accessor*.
- X in the set column indicates the attribute *must have a `set` accessor*.

Testing

The exercise tests have been organized into three categories.

The first category of tests check the structure of a class. If it's a subclass, does it extend the correct superclass? Are the correct constructors in place? What about the fields and their getters and setters, are they present in code? Finally, are the required methods and their parameters at least defined?

The second category tests the happy path behaviors of a class. Do the constructors set the fields they're supposed to? Do the methods produce the correct results?

The last category checks the edge case behaviors of a class. Are default values correctly set? Do methods handle boundary values?

The tests in the second and third categories are ignored until all the tests in the first category pass. That is, you'll need to create a well-formed class before the tests in the other categories are even run. Similarly, the edge case tests are ignored until the happy path tests pass. **Because the three categories are dependent upon one another, you need to run the tests on the class-level rather than individually running a test as you might normally do.**

You don't need to successfully pass all three categories of the `BankAccount` tests before moving onto either the `CheckingAccount` or `SavingsAccount`, but you're strongly encouraged to successfully pass at least the happy path behavioral tests for the `BankAccount` class given the other two are subclasses of it.

Finally, beyond saying the balance defaults to 0 and the constructors set the `accountHolderName` and `accountNumber` fields, the `BankAccount` specification is silent regarding the *state* of the class. For instance, there's no prohibition on initializing an account with balance less than 0, or depositing a negative amount. This is intentional. The exercise is on inheritance and overriding, *the tests only confirm the behaviors actually described in any of the specifications*.

Instructions

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

Step One: Implement the `BankAccount` class

The `BankAccount` class represents a basic checking or savings account at a bank.

Constructor	Description
BankAccount(String accountHolderName, String accountNumber)	A new bank account requires an account holder name and account number. The balance defaults to a 0 dollar balance.
BankAccount(String accountHolderName, String accountNumber, int balance)	A new bank account requires an account holder name and account number. The balance is initialized to the dollar balance given.

Attribute Name	Data Type	Get	Set	Description
accountHolderName	String	X		Returns the account holder name that the account belongs to.
accountNumber	String	X		Returns the account number that the account belongs to.
balance	int	X		Returns the balance value of the bank account in dollars.

Method Name	Return Type	Description
deposit(int amountToDeposit)	int	Adds `amountToDeposit` to the current balance, and returns the new balance of the bank account.

Method Name	Return Type	Description
withdraw(int amountToWithdraw)	int	Subtracts `amountToWithdraw` from the current balance, and returns the new balance of the bank account.

Step Two: Implement the `CheckingAccount` class

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

Override Method	Description
withdraw	If the balance falls below $0.00 but is still greater than -$100.00, a $10.00 overdraft fee is also charged against the account.
	A request to overdraw a checking account by $100.00 or more (before overdraft fee) fails and the balance remains the same. For example, if the current balance is -$89.00, and the amount to withdraw is $10.00, the resulting balance is -$99.00. The withdrawal is permitted since the new balance is greater than -$100.00. The $10.00 overdraft fee is then charged against the account, resulting in a final balance of -$109.00. A withdrawal of $11.00 in the same situation fails because the new balance would be -$100.00 which is equal to, *not greater than*, the lower limit of -$100.00.

Step Three: Implement the `SavingsAccount` class

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

Override Method	Description
withdraw	If the remaining balance is less than $150.00 after a withdrawal is made, an additional $2.00 service charge is withdrawn from the account.
	If a withdrawal is requested that would result in a negative balance (including the service charge), the withdrawal fails and balance remains the same. No fees are incurred.

Sample usage

```
BankAccount checkingAccount = new CheckingAccount("Bernice", "CHK:1234");
BankAccount savingsAccount = new SavingsAccount("Bernice", "SAV:9876");

int amountToDeposit = 2;
int newBalance = checkingAccount.deposit(amountToDeposit);
```

Challenge

The industry standard way to deal with decimal numbers in Java is with the `BigDecimal` class.

Can you add a new implementation for `BankAccount`, `CheckingAccount`, and `SavingsAccount` that uses `BigDecimal` for the `getBalance`, `withdraw`, and `deposit` methods instead of `int`? To do this, create a new package within `src/main/java` and name it `com.techelevator.challenge`.

There are no tests associated with the challenge. Be sure to submit your work using `int` in the `BankAccount`, `CheckingAccount`, and `SavingsAccount` classes.

Tips and tricks

- A good way to determine if you're 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 helps you to quickly identify opportunities to improve your code.

- derived-properties
- inheritance-and-an-is-a-relationship