Ch. 8 CSE3130 - Object Oriented Programming 2: Reflection Log

Zephram Gilson

UEmployee, Faculty, Staff (With client code UniEmployee)

This program utilizes a **Faculty** class and **Staff** class, both extended from the **UEmployee class**, to manage university employee information such as name, salary, department (for faculty), and/or job title (for staff).

One of the changes I made in my coding process was in my usage of variable names. In my constructors, I was using different variable names like `employeeName` and `employeeSalary` to take in the employee's name and salary.

```
public class UEmployee {
    private String name;
    private double salary;

    // Constructor
    public UEmployee(String employeeName, double employeeSalary) {
        employeeName = name;
        employeeSalary = salary;
    }
```

I found that it was more efficient and clear to use the same variable names as the class's instance variables (i.e., `name` and `salary`) and differentiate them using `this.`

```
public class UEmployee {
    private String name;
    private double salary;

    // Constructor
    public UEmployee(String name, double salary) {
        this.name = name;
        this.salary = salary;
    }
```

By doing this, I eliminated the need for separate local variable names and directly assigned the parameters to the instance variables, making the code cleaner and easier to maintain. I made this change for all my constructors.

Apart from that, my UEmployee class includes:

```
    // Getter for name
    public String getName() {
        return name;
    }

    // Getter for salary
    public double getSalary() {
        return salary;
    }

    // toString method for easy display of employee information
    @Override
    public String toString() {
        return "Employee Name: " + name + ", Salary: $" + salary;
    }
```

- A getter for the name of the employee
- A getter for the salary of the employee
- A toString method for display of employee information.

Then, the Faculty class (extended from UEmployee) first declares the private string `department,` then includes a constructor for the faculty member, which then uses `super(name, salary)` to call the constructor of UEmployee. The faculty class also includes a getter for department, and overrides the toString to include department info:

```
public class Faculty extends UEmployee {
    private String department;

    // Constructor
    public Faculty(String name, double salary, String department) {
        super(name, salary); // Calls the constructor of UEmployee
        this.department = department;
    }

    // Getter for department
    public String getDepartment() {
        return department;
    }

    // Override toString to include department info
    @Override
    public String toString() {
        return super.toString() + ", Department: " + department;
    }
}
```

Similarly, my **Staff** class (also extended from **UEmployee**) works the same as the **Faculty** class, however instead of getting the department name and overriding the **toString** to display department info, it overrides the **toString** to include job title info:

```
public class Staff extends UEmployee {
    private String jobTitle;

    // Constructor
    public Staff(String name, double salary, String jobTitle) {
        super(name, salary); // Calls the constructor of UEmployee
        this.jobTitle = jobTitle;
    }

    // Getter for job title
    public String getJobTitle() {
        return jobTitle;
    }

    // Override toString to include job title info
    @Override
    public String toString() {
        return super.toString() + ", Job Title: " + jobTitle;
    }
}
```

Lastly, all the code is tested with client code **University:**

```java
package mastery.UniEmployees;

public class University {
    public static void main(String[] args) {

        // Create a faculty object
        Faculty faculty = new Faculty("Dr. Gilson", 85000, "Computer Science");
        System.out.println(faculty);

        // Create and print a staff object
        Staff staff = new Staff("John Doe", 45000, "Administrative Assistant");
        System.out.print(staff);

    }
}
```

The only changes I made in my process were the changes in the variables, which made my code more readable to me and more efficient.

Vehicle, Car, Truck, Minivan (With client code CreateVehicle)

This program utilizes the **Car**, **Truck**, and **Minivan** classes, all extended from the **Vehicle** class, to manage information such as fuel economy, seating capacity, and cargo volume for different types of vehicles. Each subclass has its own specific characteristics, such as body style for cars, towing capacity for trucks, and rear entertainment systems for minivans.

```java
public abstract class Vehicle {
    private double fuelEconomyCity; // Fuel economy in city (mpg)
    private double fuelEconomyHwy;  // Fuel economy on highway (mpg)
    private int seatingCapacity;    // Number of seats
    private double cargoVolume;     // Cargo volume (cubic feet)

    // Constructor
    public Vehicle(double fuelEconomyCity, double fuelEconomyHwy, int seatingCapacity, double cargoVolume) {
        this.fuelEconomyCity = fuelEconomyCity;
        this.fuelEconomyHwy = fuelEconomyHwy;
        this.seatingCapacity = seatingCapacity;
        this.cargoVolume = cargoVolume;
    }

    // Abstract method to be implemented by subclasses
    public abstract void displayVehicleType();
```

First, in my Vehicle class, I added a base constructor for a vehicle that takes in the variables `fuelEconomyCity`, `fuelEconomyHwy`, `seatingCapacity`, and `cargoVolume`. These parameters cover general characteristics that are common across different vehicle types, like how fuel-efficient the vehicle is in both city and highway driving, the number of seats it can hold, and its cargo capacity in cubic feet.

To keep each value associated with an instance, I used `this` to assign each parameter to the corresponding instance variable (`fuelEconomyCity`, `fuelEconomyHwy`, `seatingCapacity`, `cargoVolume`). This ensures each instance of `Vehicle` holds its own values for these properties. (This was a change I made, similar to in my first project.)

(Before using `this.` - constructor uses different variable names, which I saw as less efficient)

```
// Constructor
public Vehicle(double fEconomyCity, double fEconomyHwy, int stgCapacity, double cgVolume) {
    fEconomyCity = fuelEconomyCity;
    fEconomyHwy = fuelEconomyHwy;
    stgCapacity = seatingCapacity;
    cgVolume = cargoVolume;
}
```

Next, I added getter methods for each of the instance variables: `fuelEconomyCity, fuelEconomyHwy, seatingCapacity,` and `cargoVolume.` These methods allow access to each private variable from outside the class, following encapsulation principles.

```
// Getter methods
public double getFuelEconomyCity() {
    return fuelEconomyCity;
}

public double getFuelEconomyHwy() {
    return fuelEconomyHwy;
}

public int getSeatingCapacity() {
    return seatingCapacity;
}

public double getCargoVolume() {
    return cargoVolume;
}
```

Lastly, in the **Vehicle** class, I added a `toString` method to display these details in each vehicle.

```
// toString method to display common details
@Override
public String toString() {
    return "Fuel Economy (City): " + fuelEconomyCity + " mpg, " +
           "Fuel Economy (Highway): " + fuelEconomyHwy + " mpg, " +
           "Seating Capacity: " + seatingCapacity + ", " +
           "Cargo Volume: " + cargoVolume + " cubic feet";
}
```

Next, I created my 3 subclasses, **Car, Minivan,** and **Truck.**

Each of the subclasses have their own unique additional member:

For **Car:**

```
public class Car extends Vehicle {
    private String bodyStyle;  // Additional member for a car
```

`bodyStyle;` (ex. sedan, compact, wagon, etc)

For **Minivan:**

```
public class Minivan extends Vehicle {
    private boolean hasRearEntertainmentSystem;  // Additional member for minivan
```

`hasRearviewEntertainmentSystem;` **(**a boolean describing whether the minivan has an entertainment system in the back)

And for **Truck:**

```
public class Truck extends Vehicle {
    private double towingCapacity;  // Additional member for truck
```

`towingCapacity;` (a quantitative value of the towing capacity of the truck)

Each of the classes also include a toString method displaying their respective unique additional member:

```
  // toString method to display VEHICLE-specific details
  @Override
  public String toString() {
    return super.toString() + ", <Unique Member>*: " + <Variable of Unique Member>*;
  }
```

***** (<Unique Member> is either “Body Style: ” or “Towing Capacity: ” or “Rear Entertainment System: ” depending on the class, and <Variable of Unique Member> is the respective variable depending on the class)

All of the classes also each have an additional boolean method specific to their unique member, and an implementation of the abstract method.

(Example from Minivan class)

```
  // Implement abstract method
  @Override
  public void displayVehicleType() {
      System.out.println("This is a Minivan.");
  }
  // Additional method specific to Minivan
  public boolean hasRearEntertainmentSystem() {
      return hasRearEntertainmentSystem;
  }
```

The second method, `public boolean hasRearEntertainmentSystem()` is an additional feature to return whether or not the van has a rear entertainment system, the truck class has a `public double getTowingCapacity()` to get and display the towing capacity of the truck, and same for the Car class with its unique body type member.

Finally, I created client code to test my program:

```java
public class CreateVehicle {
    public static void main(String[] args) {
        // Create a Car object
        Car car = new Car(25.0, 35.0, 5, 15.0, "Sedan");
        car.displayVehicleType();
        System.out.println(car);

        // Create a Truck object
        Truck truck = new Truck(15.0, 20.0, 3, 30.0, 10000);
        truck.displayVehicleType();
        System.out.println(truck);

        // Create a Minivan object
        Minivan minivan = new Minivan(20.0, 28.0, 7, 40.0, true);
        minivan.displayVehicleType();
        System.out.println(minivan);
    }
}
```

Output:

```
This is a Car.
Fuel Economy (City): 25.0 mpg, Fuel Economy (Highway): 35.0 mpg, Seating Capacity: 5, Cargo Volume: 15.0 cubic feet, Body Style: Sedan
This is a Truck.
Fuel Economy (City): 15.0 mpg, Fuel Economy (Highway): 20.0 mpg, Seating Capacity: 3, Cargo Volume: 30.0 cubic feet, Towing Capacity: 10000.0 lbs
This is a Minivan.
Fuel Economy (City): 20.0 mpg, Fuel Economy (Highway): 28.0 mpg, Seating Capacity: 7, Cargo Volume: 40.0 cubic feet, Rear Entertainment System: Yes
```

I made no other major changes to my program throughout my process.

Account, PersonalAcct, BusinessAcct (With client code AccountsRun)

This program uses the **Account** class as a foundation to manage customer accounts, including basic details such as balance and customer information. Through inheritance, the **PersonalAcct** and **BusinessAcct** subclasses add specific functionality for personal and business accounts, including minimum balance requirements and associated penalty fees. Additionally, the **Customer** class holds customer address information, which can be updated through the **Account** class. The client code **AccountsRun** demonstrates account creation, withdrawal processing, and the application of penalties for balances below the minimum threshold for each account type.

In my **Customer** class, I created all of the modifier methods relating to changing account location:

```java
public class Customer {
    {
        firstName = fName;
        lastName = lName;

        //reflect the changes in the parameter
    }


    //create changeCity method that asks the user their city and records city in a variable above

    //create changeStreet method that asks the user their street and records street in a variable abov

    //create changeProvince method that asks the user their province and records province in a variabl

    //create changePostalCode method that asks the user their postal code and records postal code in a
```

the completed `changeCity, changeStreet, changeProvince,` and `changePostalCode` modifier methods look like:

```java
/**
 * Updates the customer's city.
 * pre: none
 * post: The customer's city has been updated.
 */
public void changeCity(String city) {
    this.city = city;
}

/**
 * Updates the customer's street.
 * pre: none
 * post: The customer's street has been updated.
 */
public void changeStreet(String street) {
    this.street = street;
}

/**
 * Updates the customer's province.
 * pre: none
 * post: The customer's province has been updated.
 */
public void changeProvince(String province) {
    this.province = province;
}

/**
 * Updates the customer's postal code.
 * pre: none
 * post: The customer's postal code has been updated.
 */
public void changePostalCode(String postalCode) {
    this.postalCode = postalCode;
```

They all have no pre-conditions and, as a post, update the customer's respective location information.

Lastly, in the **Customer** class, the `toString` method returns a string that represents the Customer object.

```java
/**
 * Returns a String that represents the Customer object.
 * pre: none
 * post: A string representing the Customer object has been returned.
 */
@Override
public String toString() {
    // Updated to include full address details
    return firstName + " " + lastName + "\n" + street + ", " + city + ", " + province + " " + postalCode + "\n";
}
```

Next, in the **Account** class, I created a `changeAddress` method that calls the `cust` object from above in order to change:

```java
/**
 * Changes the address for the customer.
 * pre: none
 * post: The customer's address has been updated.
 */
public void changeAddress(String street, String city, String province, String postalCode) {
    cust.changeStreet(street);
    cust.changeCity(city);
    cust.changeProvince(province);
    cust.changePostalCode(postalCode);
}
```

Other than that, the **Account** class includes:

- Constructors to create the account

```java
/**
 * Constructor with initial balance and customer details including address
 * pre: none
 * post: An account has been created. Balance and customer data have been initialized.
 */
public Account(double bal, String fName, String lName, String street, String city, String province, String postalCode) {
    balance = bal;
    cust = new Customer(fName, lName, street, city, province, postalCode);
    acctID = fName.substring(0, 1) + lName;
}

/**
 * Constructor with only account ID
 * pre: none
 * post: An empty account has been created with the specified account ID.
 */
public Account(String ID) {
    balance = 0;
    cust = new Customer("", "", "", "", "", "");
    acctID = ID;
}
```

- The `getBalance, deposit, withdrawal,` etc. methods

```java
/**
 * Returns the account ID.
 * pre: none
 * post: The account ID has been returned.
 */
public String getID() {
    return acctID;
}

/**
 * Returns the current balance.
 * pre: none
 * post: The account balance has been returned.
 */
public double getBalance() {
    return balance;
}

/**
 * Makes a deposit to the account.
 * pre: none
 * post: The balance has been increased by the amount of the deposit.
 */
public void deposit(double amt) {
    balance += amt;
}

/**
 * Makes a withdrawal from the account if there is enough money.
 * pre: none
 * post: The balance has been decreased by the amount withdrawn.
 */
public void withdrawal(double amt) {
    if (amt <= balance) {
        balance -= amt;
    } else {
        System.out.println("Not enough money in account.");
    }
}
```

- A `boolean` to return `true` if accounts have matching IDs

```java
/**
 * Returns true if the account IDs match.
 * pre: none
 * post: true has been returned when the objects are equal, false otherwise.
 */
public boolean equals(Object acct) {
    if (acct instanceof Account) {
        Account testAcct = (Account) acct;
        return acctID.equals(testAcct.acctID);
    }
    return false;
}

/**
```

- And the `toString` method to return a string representing the **Account** object.

```java
/**
 * Returns a string representing the Account object.
 * pre: none
 * post: A string representing the Account object has been returned.
 */
@Override
public String toString() {
    NumberFormat money = NumberFormat.getCurrencyInstance();
    String accountString = "Account ID: " + acctID + "\n";
    accountString += cust.toString();
    accountString += "Current balance is " + money.format(balance) + "\n";
    return accountString;
}
```

For the **PersonalAcct** class, I started by defining the class `PersonalAcct` as a subclass of `Account` and declared my variables:

```java
public class PersonalAcct extends Account {
    private static final double MIN_BALANCE = 100.0;
    private static final double PENALTY = 2.0;
```

These variables specify the minimum balance requirement and penalty amount, which are specific to personal accounts.

Next, I added a constructor which takes the starting balance, name, and address details, and then calls the `super()` constructor of the `Account` class to initialize these values. This means `PersonalAcct` inherits the standard initialization process of `Account`, keeping it consistent with other account types while adding a unique minimum balance and penalty.

```java
public PersonalAcct(double bal, String fName, String lName, String street, String city, String province, String postalCode) {
    super(bal, fName, lName, street, city, province, postalCode);
}
```

Next, I focused on how `PersonalAcct` should handle withdrawals differently. In this account type, a penalty should apply if a withdrawal causes the balance to fall below the minimum. To achieve this, I overrode the `withdrawal(double amt)` method from `Account`.

```
@Override
public void withdrawal(double amt) {
    super.withdrawal(amt);
    if (getBalance() < MIN_BALANCE) {
        super.withdrawal(PENALTY);
        System.out.println("Balance below minimum. Penalty of $" + PENALTY + " applied.");
    }
}
```

By calling `super.withdrawal(amt);`, I ensure the standard withdrawal process is executed first, including any checks or balance adjustments provided by `Account`. `PersonalAcct` only adds extra behavior for penalties instead of rewriting the entire withdrawal process.

After the withdrawal is made, I use `getBalance()` (inherited from `Account`) to see if the current balance falls below the minimum threshold defined by `MIN_BALANCE.`

If the balance is below `MIN_BALANCE,` I call `super.withdrawal(PENALTY);`, which deducts the penalty from the account balance.

To make it clear to the user that a penalty has been applied, I added a `System.out.println` message. This message provides context on the balance drop due to the penalty.

```
Withdrawing $200 from Business Account...
Balance below minimum. Penalty of $10.0 applied.
```

The **BusinessAcct** class follows the exact same process; the only difference is in the values of `MIN_BALANCE` and `PENALTY.`

```
public class BusinessAcct extends Account {
    private static final double MIN_BALANCE = 500.0;
    private static final double PENALTY = 10.0;
```

All my code can be tested simply by creating (an) object(s) with a balance above the minimum or below:

```
public static void main(String[] args) {
    // Test Personal Account with minimum balance penalty
    PersonalAcct personalAccount = new PersonalAcct(150.0, "John", "Doe", "123 Main St", "Springfield", "IL", "62704");
    System.out.println("=== Personal Account Details ===");
    System.out.println(personalAccount);
```

However I wanted to test it further, to see how the program runs when some money is withdrawn from either of the accounts such that the updated balance is below the minimum balance. So, I adjusted my code to include withdrawals:

```java
public class AccountsRun {
    public static void main(String[] args) {
        // Test Personal Account with minimum balance penalty
        PersonalAcct personalAccount = new PersonalAcct(150.0, "John", "Doe", "123 Main St", "Springfield", "IL", "62704");
        System.out.println("=== Personal Account Details ===");
        System.out.println(personalAccount);

        // Test withdraw function
        System.out.println("\nWithdrawing $60 from Personal Account...");
        personalAccount.withdrawal(60); // Should apply penalty if below minimum balance
        System.out.println(personalAccount);

        // Test Business Account with minimum balance penalty
        BusinessAcct businessAccount = new BusinessAcct(600.0, "Jane", "Smith", "456 Elm St", "Shelbyville", "CA", "62565");
        System.out.println("\n=== Business Account Details ===");
        System.out.println(businessAccount);

        // Test withdraw function
        System.out.println("\nWithdrawing $200 from Business Account...");
        businessAccount.withdrawal(200); // Should apply penalty if below minimum balance
        System.out.println(businessAccount);
    }
}
```

As a result, my output looks like this:

```
Account ID: JDoe
John Doe
123 Main St, Springfield, IL 62704
Current balance is $150.00


Withdrawing $60 from Personal Account...
Balance below minimum. Penalty of $2.0 applied.
Account ID: JDoe
John Doe
123 Main St, Springfield, IL 62704
Current balance is $88.00


=== Business Account Details ===
Account ID: JSmith
Jane Smith
456 Elm St, Shelbyville, CA 62565
Current balance is $600.00


Withdrawing $200 from Business Account...
Balance below minimum. Penalty of $10.0 applied.
Account ID: JSmith
Jane Smith
456 Elm St, Shelbyville, CA 62565
Current balance is $390.00
```

These were the only major changes I made to my code throughout my process.