CMM024 Object Oriented Programming

Practical Session: 10

This lab is dedicated to Object Orientated Programming; focusing on inheritance with a strong focus on Abstract classes and Enums. You are going to create your own classes and test them

1. Task 1

Brief

A company wants to experiment with a simple easy way to apply discount to their customers. There are two types of customers; one is the typical customers (like yourself), and the other is companies (remember that companies do need to acquire products too).

The company in question want to promote trade with other companies by providing higher rates of discount. In addition, they want 4 levels of discount.

How to use Enums effectively

- Create a folder for Lab10
- Create a Lab10.java file
- Create a main method

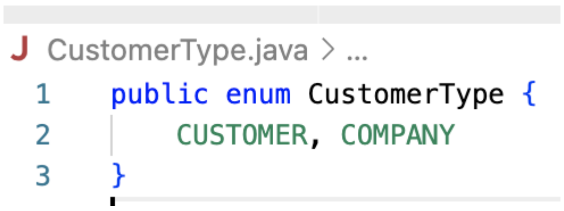
Version 1

Declare these variables and assign their respective value.

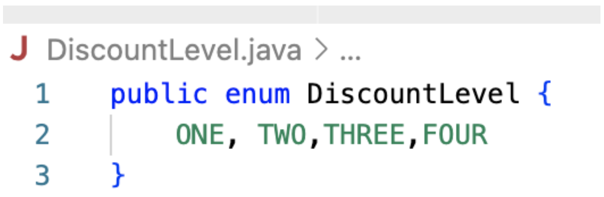
The first is a test variable assigned a price without discount. The second one is a scaling factor to apply to the original cost taking into account if the customer is a company or a actual consumer, and the discount level which goes from 1 to 4; 1being the lowest level of discount and 4 the highest.

```java
double originalPrice = 20.0;
double discountMultiplier = 0.3; //30%
```

Now we need to define some constants to hold the types we mentioned above. So let create 2 Enums as shown below


CustomerType.java > ...

```java
public enum CustomerType {
    CUSTOMER, COMPANY
}
```


DiscountLevel.java > ...

```java
public enum DiscountLevel {
    ONE, TWO,THREE,FOUR
}
```

Going back to the main method we can now declare these 2 Enums and assigned them a constant value

```java
CustomerType customer = CustomerType.COMPANY;
DiscountLevel discountLevel = DiscountLevel.FOUR;
```

As I have mentioned during the lecture, the constant are addressable using the index, i.e. the order at which they appear in the enum. The index start from 0. So, for DiscountLevel

Enum Constant	Index value
ONE	0
TWO	1
THREE	2
FOUR	3

Note: To get the index for a selected enum constant, we use the ordinal() method of the Enum.

As CustomerType is setup to COMPANY and this constant is the second one, therefore the index will be one. The DiscountLevel is set to FOUR, therefore the index value will be 3 (i.e. 0,1,2,3). Another point to make is that we need to add 1 to these indexes. The reason for this is that anything multiplied by 0 equals 0. (3 * 0 = 0). This is not what we want. By adding 1, we have (1 * 4 = 4), which is much better

$$totalDiscountLevel - (Customer\ index + 1) * (Discount\ Level + 1)$$

In terms of code, we have this

```java
double totalDiscountLevel =  (customer.ordinal()+1) * (discountLevel.ordinal()+1);
```

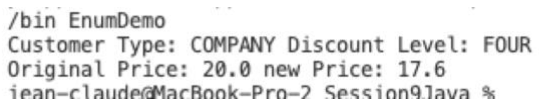
Now we have our discount level and we now the discount multiplier that is applied to it we can simply multiply them together to get total discount, which when subtracted from the original price gives us the final cost.

```java
double newCost  = originalPrice  - (totalDiscountLevel* discountMultiplier);
```

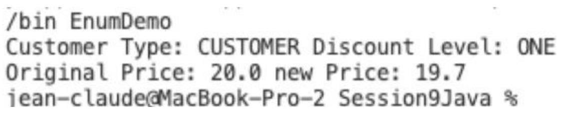
We can now write the code to display the information

```java
System.out.println("Customer Type: " + customer.toString() + " Discount Level: " + discountLevel);
System.out.println("Original Price: " + originalPrice + " new Price: " + newCost);
```

Output



```
/bin EnumDemo
Customer Type: COMPANY Discount Level: FOUR
Original Price: 20.0 new Price: 17.6
jean-claude@MacBook-Pro-2 Session9Java %
```



```
/bin EnumDemo
Customer Type: CUSTOMER Discount Level: ONE
Original Price: 20.0 new Price: 19.7
jean-claude@MacBook-Pro-2 Session9Java %
```

So, we can conclude that using the ordinal() method to retrieve the index can be useful in algorithms. It is simple but it works

Version 2

In the last example, we simply use the index returned by the ordinal() method to calculate a discount level, which when multiplied by a given ration provides the total discount. The ratio i.e. percent, is the same i.e. 0.3

What if we want the discount percent to be different for each level

Enum Constant	Discount percent for level
ONE	0.1 (10%)
TWO	0.2 (20%)
THREE	0.3 (30%)
FOUR	0.4 (40%)

Since the ordinal() method return an index starting at 0, we can use an array to store these percentages

```java
double[] discountValues = {0.1, 0.2, 0.3, 0.4};
```

And then we can address this array using the index returned by the ordinal() method!

- Change the code to use this array

```
double baseDiscountPercent = discountValues[discountLevel.ordinal()];
```

- Run the code.

Version 3

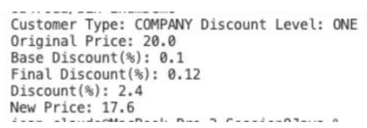
- Perform the same changes by using an array as shown below

```
double[] custumerDiscountValues = {1, 1.2}; //customer:  NOTHING, company: 20%
```

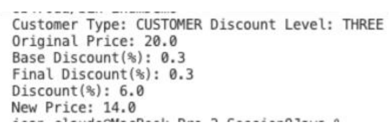
- And therefore this

```
double finalDiscountPercent =  baseDiscountPercent * (custumerDiscountValues[customerType.ordinal()]);
```

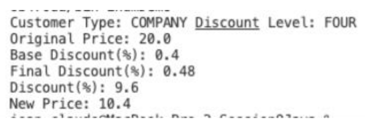
- Run the code and use different customer type and discount level as show below



```
Customer Type: COMPANY Discount Level: ONE
Original Price: 20.0
Base Discount(%): 0.1
Final Discount(%): 0.12
Discount(%): 2.4
New Price: 17.6
```



```
Customer Type: CUSTOMER Discount Level: THREE
Original Price: 20.0
Base Discount(%): 0.3
Final Discount(%): 0.3
Discount(%): 6.0
New Price: 14.0
```

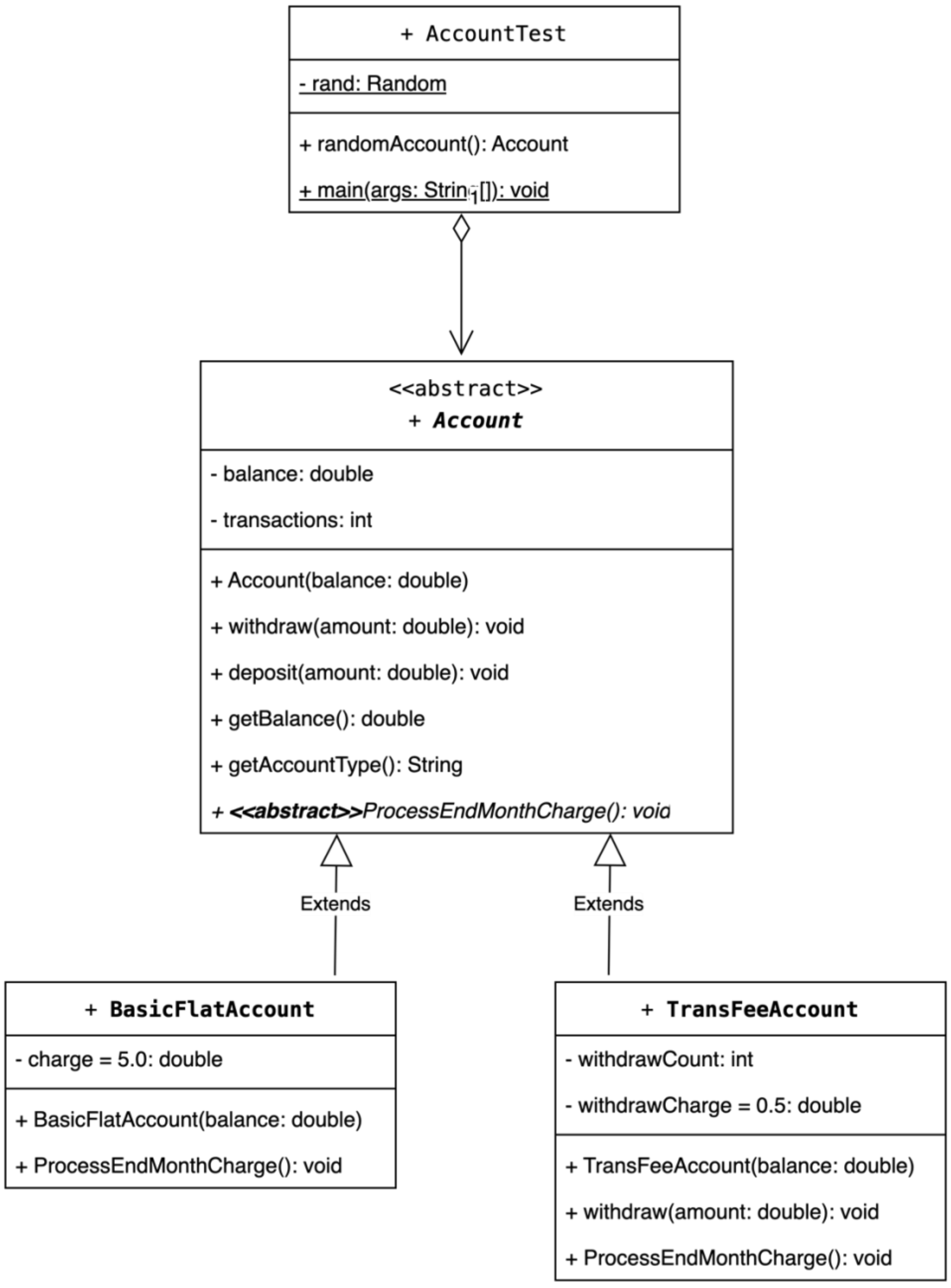


```
Customer Type: COMPANY Discount Level: FOUR
Original Price: 20.0
Base Discount(%): 0.4
Final Discount(%): 0.48
Discount(%): 9.6
New Price: 10.4
```

2. Task 2

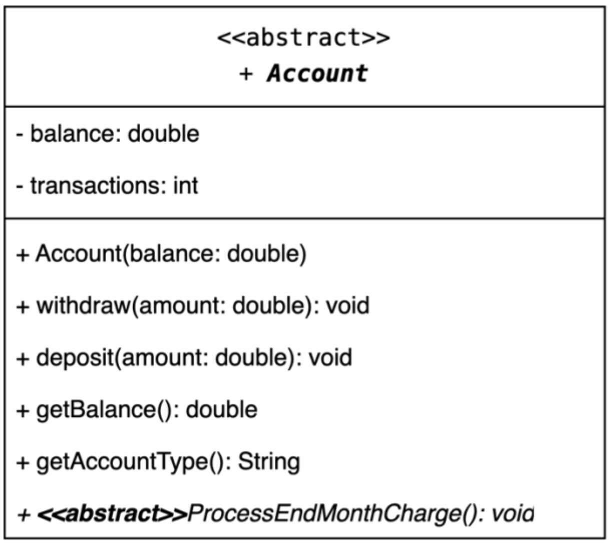
What is interesting in this exercise is the fact that the subclasses invoke the superclass methods as we as the superclass invoking subclass overridden methods.

The second interesting point here is that one subclass overrides the withdraw method of the superclass so it can perform a more specialised calculations when it comes to calculating the charges that will be withdrawn from the account.



+ AccountTest
- rand: Random
+ randomAccount(): Account
+ main(args: String[]): void
<<abstract>>
+ Account
- balance: double
- transactions: int
+ Account(balance: double)
+ withdraw(amount: double): void
+ deposit(amount: double): void
+ getBalance(): double
+ getAccountType(): String
+ <<abstract>>ProcessEndMonthCharge(): void
Extends
Extends
+ BasicFlatAccount
- charge = 5.0: double
+ BasicFlatAccount(balance: double)
+ ProcessEndMonthCharge(): void
+ TransFeeAccount
- withdrawCount: int
- withdrawCharge = 0.5: double
+ TransFeeAccount(balance: double)
+ withdraw(amount: double): void
+ ProcessEndMonthCharge(): void

Account


<<abstract>>
+ Account
- balance: double
- transactions: int
+ Account(balance: double)
+ withdraw(amount: double): void
+ deposit(amount: double): void
+ getBalance(): double
+ getAccountType(): String
+ <<abstract>>ProcessEndMonthCharge(): void

Constructor(...)

- Initialise the balance field
- Set transactions to 0

Withdraw(...)

- Decrease amount from balance
- increase transactions by 1

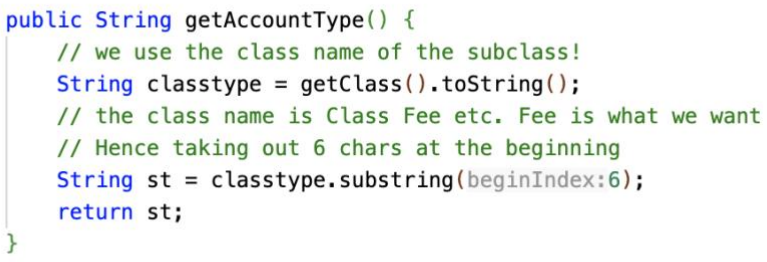
deposit(...)

- Add amount to balance
- increase transactions by 1

getBalance(...)

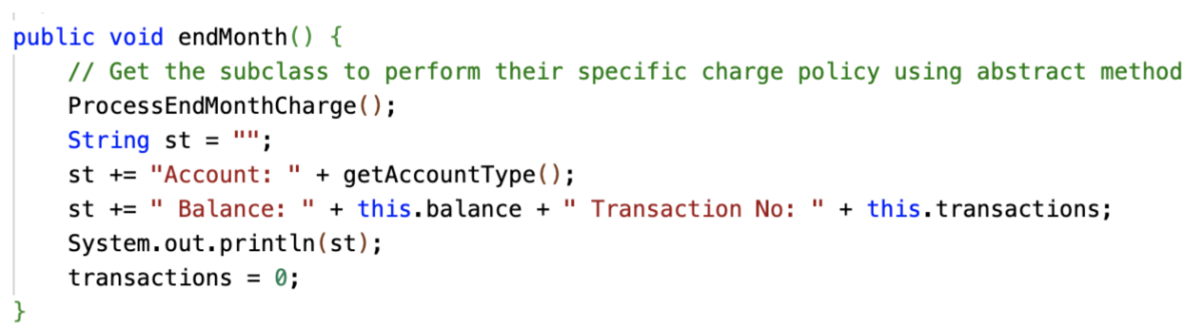
- return the value of balance field

getAccountType(...)



```java
public String getAccountType() {
    // we use the class name of the subclass!
    String classtype = getClass().toString();
    // the class name is Class Fee etc. Fee is what we want
    // Hence taking out 6 chars at the beginning
    String st = classtype.substring(beginIndex:6);
    return st;
}
```

endMonth(...)

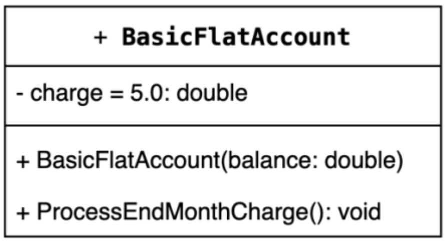


```java
public void endMonth() {
    // Get the subclass to perform their specific charge policy using abstract method
    ProcessEndMonthCharge();
    String st = "";
    st += "Account: " + getAccountType();
    st += " Balance: " + this.balance + " Transaction No: " + this.transactions;
    System.out.println(st);
    transactions = 0;
}
```

ProcessEndMonthCharge(...)

- abstract method. No implementation

BasicFlatAccount (extends Account)


+ BasicFlatAccount
- charge = 5.0: double
+ BasicFlatAccount(balance: double)
+ ProcessEndMonthCharge(): void

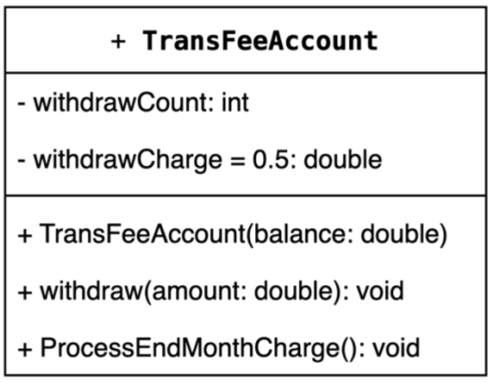
Constructor(...)

- Initialise superclass

ProcessEndMonthCharge

- Withdraw the charge of 5.0 (invoke super class withdraw method)

TransFeeAccount (extends Account)


+ TransFeeAccount
- withdrawCount: int
- withdrawCharge = 0.5: double
+ TransFeeAccount(balance: double)
+ withdraw(amount: double): void
+ ProcessEndMonthCharge(): void

Constructor(…)

- Initialise superclass
- Set withdraw count to 0

withdraw(…)

- Note: override superclass withdraw method
- Invoke superclass withdraw method passing amount
- Increase withdraw count

ProcessEndMonthCharge

- Compute charges (withdraw count * withdraw charge)
- Set withdraw count to zero

AccountTest

Note: there are two types of account. So, we can get a random number from 0 to 1, if 0 then we create a TransFeeAccount otherwise if it is 1 then we create a BasicFlatAccount account

Note2: To simulate the transactions, we also use random numbers. However, we want to bias the randomness toward depositing money. If we get a random number from 0 to 99, if we say if smaller than 20, we withdraw money, and over 20 we deposit, then the chances of depositing are 5 times more probable than withdrawing.

Note3: for the simulation, we use a for loop 1 to 31, and process transaction for each account etc.

```
import java.util.Random;


public class AccountTest {


   // Allocate a Random object shared by these static methods

   private static Random rand = new Random();

   private static final int NUM_ACCOUNTS = 5;


   // Return a new random account of a random type.
```

```
private static Account randomAccount() {
    int totalAccountType = 2;
    //pick a type at random
    int pick = rand.nextInt(totalAccountType);
    Account account = null;
    switch (pick) {
        case 0:
            account = new TransFeeAccount(rand.nextInt(100));
            break;
        case 1:
            account = new BasicFlatAccount(rand.nextInt(100));
            break;
    }
    System.out.println("Creating a random account: " + account.getAccountType());
    return (account);
}
public static void main(String args[]) {
    // Create some assorted accounts
    Account[] accounts = new Account[NUM_ACCOUNTS];
    // create different types of account at random
    System.out.println("\nCreating " + NUM_ACCOUNTS + " at ramdom\n");
    for (int i = 0; i < accounts.length; i++) {
        accounts[i] = randomAccount();
    }
    System.out.println("\nSimulating transactions for the whole months (31 days\n");
    // Simulate some transactions for each account at random
    for (int day = 1; day <= 31; day++) {
          // select a created account at random

          int accountNum = rand.nextInt(accounts.length);

          Account currentAccount =  accounts[accountNum];

          // now get an integer at random from 0 to 1

          int randomNumber = rand.nextInt(100);

          // System.out.println("Random number: " + randomNumber);

          // if random number is zero we withdraaw otherwise we deposite

           //random amount to withdraw from 1 to 100

             double amount = rand.nextInt(100) + 1;

          if (randomNumber < 20) { // do something to that account

              System.out.println(currentAccount.getAccountType() + " Withdrawing: " +
amount);

              //invoke the supclass withdraw method

             currentAccount.withdraw(amount);

          } else {

              System.out.println(currentAccount.getAccountType() +" Deposit: " +
amount);

               //invoke the supclass deposit method

             currentAccount.deposit(amount);

          }

       }

       // Display all accounts detaails

       System.out.println("\nEnd of month Information for the account...\n");

       for (int acct = 0; acct < accounts.length; acct++) {

          accounts[acct].endMonth(); // Polymorphism Yay!

       }

       System.out.println();

    }
```

```
}
```