

NAME : NANDAWULA MARIA

REG NO : S20B23/207

ACCESS NO : A90651

COURSE : BSCS

COURSE UNIT : SOFTWARE PROJECT MANAGEMENT

LECTURER : MR. SIMON FRED LUBAMBO

class Account {

float principal;

float rate;

int daysActive;

int accountType;

public static final int STANDARD = 0;

public static final int BUDGET = 1;

public static final int PREMIUM = 2;

public static final int PREMIUM\_PLUS = 3;

}

float calculateFee(Account accounts[]) {

float totalFee = 0;

Account account;

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

account = accounts[i];

if ( account.accountType == Account.PREMIUM ||

account.accountType == Account.PREMIUM\_PLUS ) {

totalFee += .0125 \* ( account.principal

\* Math.exp( account.rate \* (account.daysActive/365.25) )

- account.principal );

}

}

return totalFee;

}

Since this calculation only uses data that is available in the Account  
class, we can move it into that class, by adding a new method called interestEarned

class Account {

float principal;

float rate;

int daysActive;

int accountType;

public static final int STANDARD = 0;

public static final int BUDGET = 1;

public static final int PREMIUM = 2;

public static final int PREMIUM\_PLUS = 3;

**float interestEarned() {**

**return ( principal \* (float) Math.exp( rate \* (daysActive / 365.25 ) ) ) - principal;  
 }**

}

float calculateFee(Account accounts[]) {

float totalFee = 0;

Account account;

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

account = accounts[i];

if ( account.accountType == Account.PREMIUM ||

account.accountType == Account.PREMIUM\_PLUS ) {

totalFee += .0125 \* ( account.principal

\* Math.exp( account.rate \* (account.daysActive/365.25) )

- account.principal );

}

}

return totalFee;

}

We can use the Replace Magic Number with Symbolic Constant refactoring, replacing it with the constant BROKER\_FEE\_PERCENT and defining that constant later

class Account {

float principal;

float rate;

int daysActive;

int accountType;

public static final int STANDARD = 0;

public static final int BUDGET = 1;

public static final int PREMIUM = 2;

public static final int PREMIUM\_PLUS = 3;

**float interestEarned() {**

**return ( principal \* (float) Math.exp( rate \* (daysActive / 365.25 ) ) ) - principal;  
 }**

}

float calculateFee(Account accounts[]) {

float totalFee = 0;

Account account;

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

account = accounts[i];

if ( account.accountType == Account.PREMIUM ||

account.accountType == Account.PREMIUM\_PLUS ) {

**totalFee += BROKER\_FEE\_PERCENT \* account.interestEarned();** }

}

return totalFee;

}

**static final double BROKER\_FEE\_PERCENT = 0.0125;**

It isn’t clear why the account is only being checked for just two types yet there are several account types

By using the Decompose Conditional refactoring, we can be able to clarify the purpose of this code.  
Adding the isPremium function to the Account class can make it more obvious that  
this is a check to verify whether the account is a premium account:

class Account {  
 float principal;  
 float rate;  
 int daysActive;  
 int accountType;  
  
public static final int STANDARD = 0;  
 public static final int BUDGET = 1;  
 public static final int PREMIUM = 2;  
 public static final int PREMIUM\_PLUS = 3;  
  
 float interestEarned() {  
 return ( principal \* (float) Math.exp( rate \* (daysActive / 365.25 ) ) )  
 - principal;  
 }  
  
 **public boolean isPremium() {  
 if (accountType == Account.PREMIUM || accountType == Account.PREMIUM\_PLUS)  
 return true;  
 else  
 return false;  
 }** }  
   
 float calculateFee(Account accounts[]) {  
 float totalFee = 0;  
 Account account;  
 for (int i = 0; i < accounts.length; i++) {  
 account = accounts[i];  
 **if ( account.isPremium() )** totalFee += BROKER\_FEE\_PERCENT \* account.interestEarned();  
 }  
 return totalFee;  
 }  
  
 static final double BROKER\_FEE\_PERCENT = 0.0125;

The last problem found during the inspection involved the interestEarned() method which  
is extracted. It is a confusing calculation, with several intermediate steps crammed into  
a single line.

The first problem is that it isnt exactly clear why there is a division by 365.25 in line 13.

The refactoring Introduce Explaining Variable can be used to introduce two intermediate variables,  
years on line 13 and compoundInterest on line 14, to clarify the code:

class Account {  
 float principal;  
 float rate;

int daysActive;  
 int accountType;  
  
public static final int STANDARD = 0;  
 public static final int BUDGET = 1;  
 public static final int PREMIUM = 2;  
 public static final int PREMIUM\_PLUS = 3;  
  
 float interestEarned() {  
 **float years = daysActive / (float) 365.25;  
 float compoundInterest = principal \* (float) Math.exp( rate \* years );  
 return ( compoundInterest – principal );** }  
  
 public boolean isPremium() {  
 if (accountType == Account.PREMIUM || accountType == Account.PREMIUM\_PLUS)  
 return true;  
 else  
 return false;  
 }  
 }  
  
 float calculateFee(Account accounts[]) {  
 float totalFee = 0;  
 Account account;  
 for (int i = 0; i < accounts.length; i++) {  
 account = accounts[i];  
 if ( account.isPremium() ) {  
 totalFee += BROKER\_FEE\_PERCENT \* account.interestEarned();  
 }  
 }  
 return totalFee;  
 }  
  
 static final double BROKER\_FEE\_PERCENT = 0.0125;

**Unit testing**

test methods from a test case object called testFeeCalculation.

There are many tests that would exercise the fee calculation function

shown in the Refactoring section above. This example shows six of them.

All of them require an  
instance of the FeeCalculation class, which is set up using this setUp() function:

public FeeCalculation feeCalculation;  
public void setUp() {  
feeCalculation = new FeeCalculation();  
}

The first test simply verifies that the function has performed its calculation and has generated the right result by comparing the output to a known value, which is calculated by hand using a calculator.

public void testTypicalResults() {  
Account accounts[] = new Account[3];  
accounts[0] = new Account();  
accounts[0].principal = 35;  
accounts[0].rate = (float) .04;  
accounts[0].daysActive = 365;  
accounts[0].accountType = Account.PREMIUM;

accounts[1] = new Account();  
accounts[1].principal = 100;  
accounts[1].rate = (float) .035;  
accounts[1].daysActive = 100;  
accounts[1].accountType = Account.BUDGET;

accounts[2] = new Account();  
accounts[2].principal = 50;  
accounts[2].rate = (float) .04;  
accounts[2].daysActive = 600;  
accounts[2].accountType = Account.PREMIUM\_PLUS;

float result = feeCalculation.calculateFee(accounts);  
assertEquals(result, (float) 0.060289, (float) 0.00001)

This test passes. The call to feeCalculation() with those three accounts returns a value of  
0.060289383, which matches the value passed to assertEquals() within the specified tolerance  
of .000001. The assertion does not cause a failure, and the test case completes.

the second test checks for a set of non-premium accounts sine you may not the have expected  
feeCalculation() to receive a set of accounts that contained no premium accounts

**public void testNonPremiumAccounts() {  
Account accounts[] = new Account[2];  
accounts[0] = new Account();  
accounts[0].principal = 12;  
accounts[0].rate = (float) .025;  
accounts[0].daysActive = 100;  
accounts[0].accountType = Account.BUDGET;  
accounts[1] = new Account();  
accounts[1].principal = 50;  
accounts[1].rate = (float) .0265;  
accounts[1].daysActive = 150;  
accounts[1].accountType = Account.STANDARD;  
float result = feeCalculation.calculateFee(accounts);  
assertEquals(result, 0, 0.0001);  
}**

The expected result for this test is 0, and it passes.