This week we did a full-recitation quiz activity.

Engage A, activity 1:

Consider the following class definition:

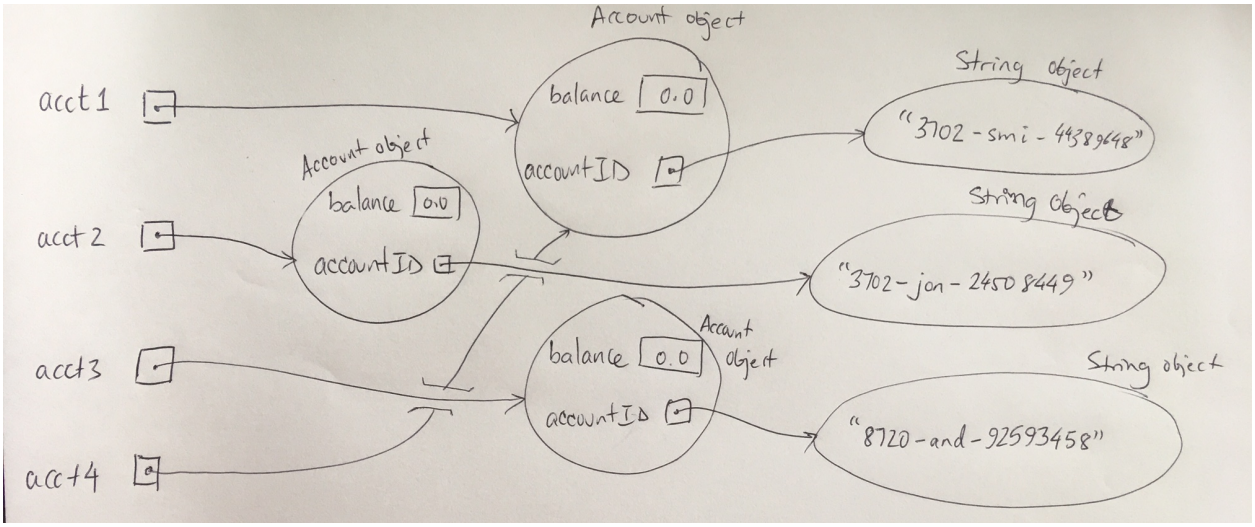
```
public class Account {
    private double balance;
    private String accountID;

    public Account(String id) {
        balance = 0.0;
        accountID = id;
    }

    public double getBalance() { return balance; }
    public String getID() { return accountID; }
    public boolean deposit(double amount) { /* definition omitted */ }
    public boolean withdraw(double amount) { /* definition omitted */ }
}
```

Draw an object diagram that shows the state of the program just after the variable acct4 is assigned a value.

```
Account acct1 = new Account("3702-smi-44389648");
Account acct2 = new Account("3702-jon-24508449");
Account acct3 = new Account("8720-and-92593458");
Account acct4 = acct1;
```

Remember that there are three things that go into an object diagram: variables (denoted by boxes, perhaps labelled by the name of the variable), references (denoted by arrows; reference arrows always start inside variable boxes and point to objects), and object (denoted by ovals). Variables can appear on their own (in which case they are local variables, and are generally drawn on the left side of a diagram) or inside object ovals (in which case they are instance variables). Objects are generally drawn on the right hand side of a diagram.

My diagram for this question in shown on the next page. Notice that because acct1 and acct4 hold the same memory address, their reference arrows both point to the same object in the diagram.



Engage A, activity 2:

Consider the following method definition for the deposit method of the Account class:

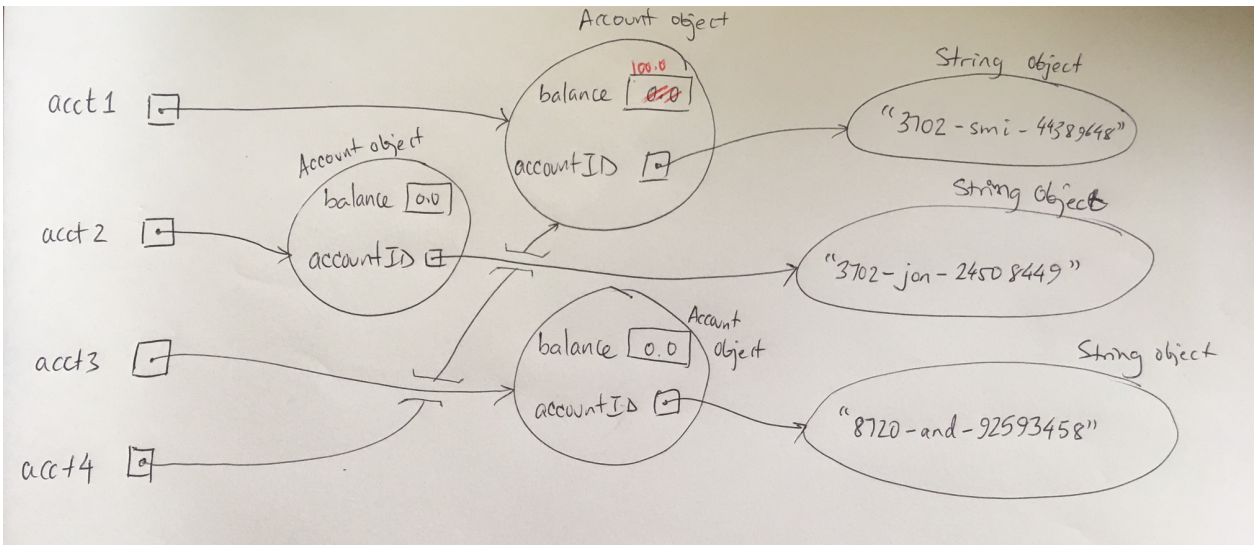
```
public boolean deposit(double amount) {
     if (amount > 0.0) {
         balance = balance + amount;
         return true;
    }
    return false;
}
```

Update your object diagram to show the state of the program after each of the following method calls:

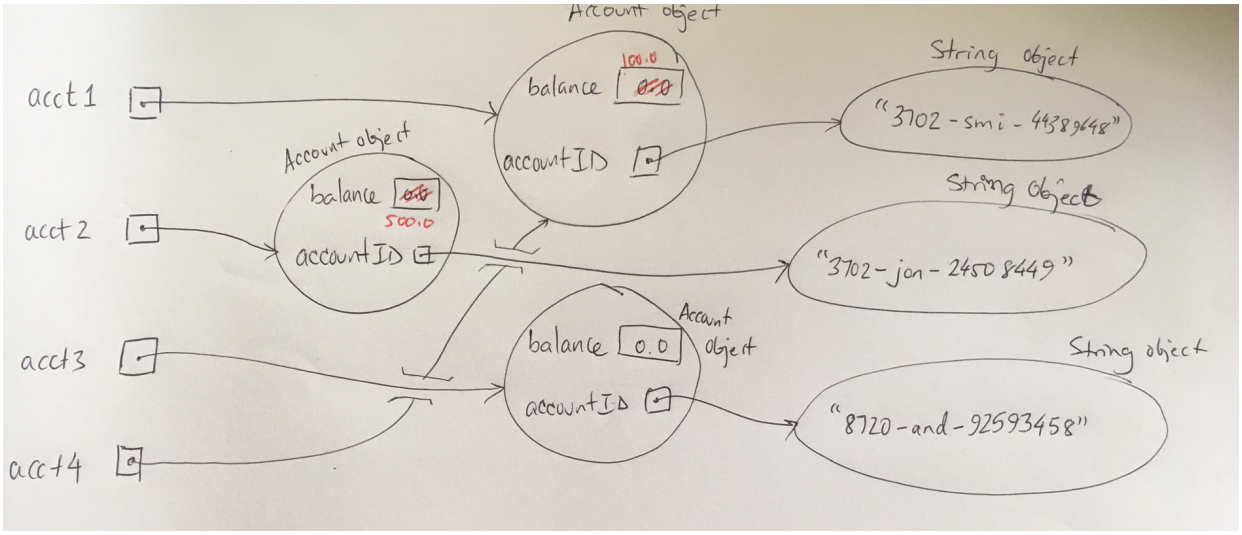
```
acct1.deposit(100.0);
acct2.deposit(500.0);
acct3.deposit(40.0);
acct4.deposit(75.0);
```

Here are my diagrams.

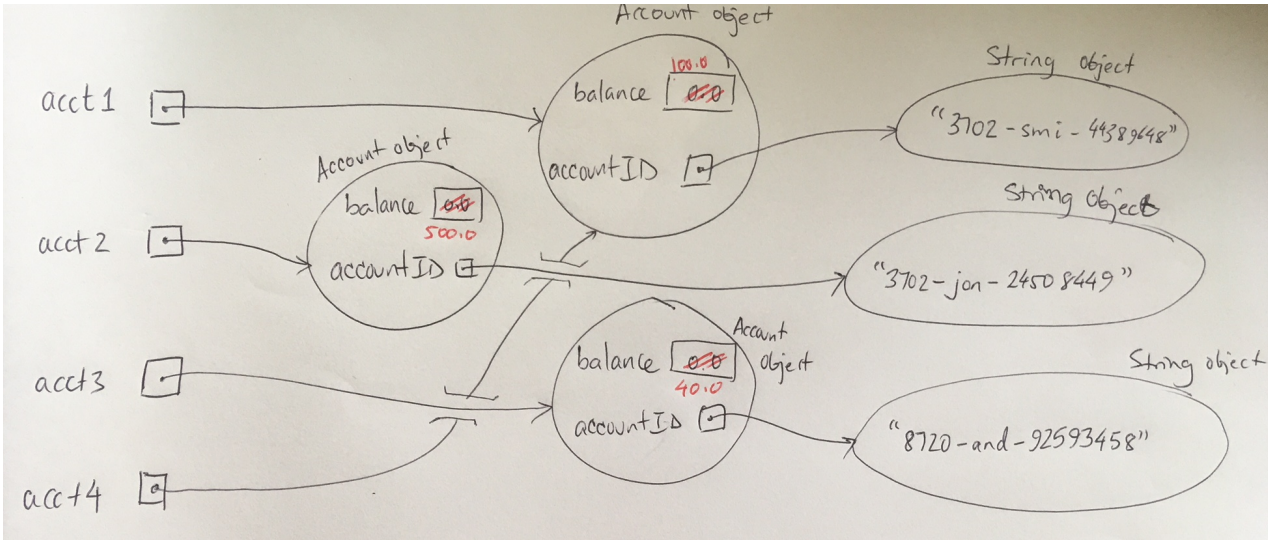
1) after `acct1.deposit(100.0);`



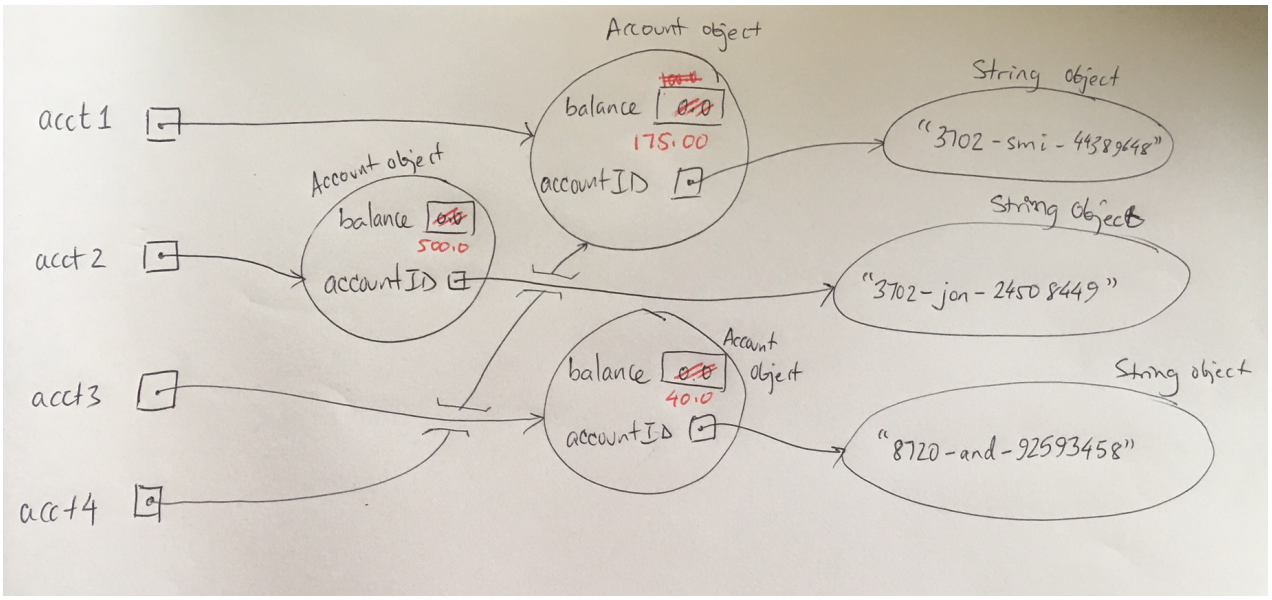
2) after `acct2.deposit(500.0);`



3) after `acct3.deposit(40.0);`



4) after `acct4.deposit(75.0);`



Engage A, activity 3:

Consider the following method definitions for the Account class:

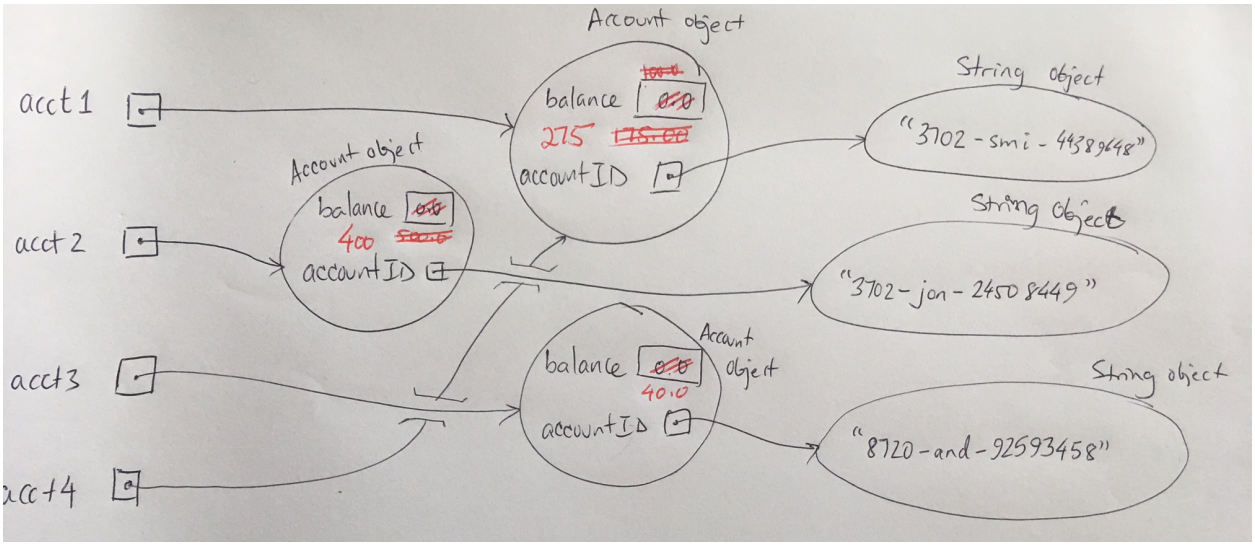
```
public boolean withdraw(double amount) {
    if (amount > 0.0 && amount <= balance) {
        balance = balance - amount;
        return true;
    }
    return false;
}

public boolean transfer(Account x, Account y, double amount) {
    if (x.withdraw(amount)) {
        y.deposit(amount);
        return true;
    }
    return false;
}
```

Update your object diagram to show the state of the program after each of the following method calls:

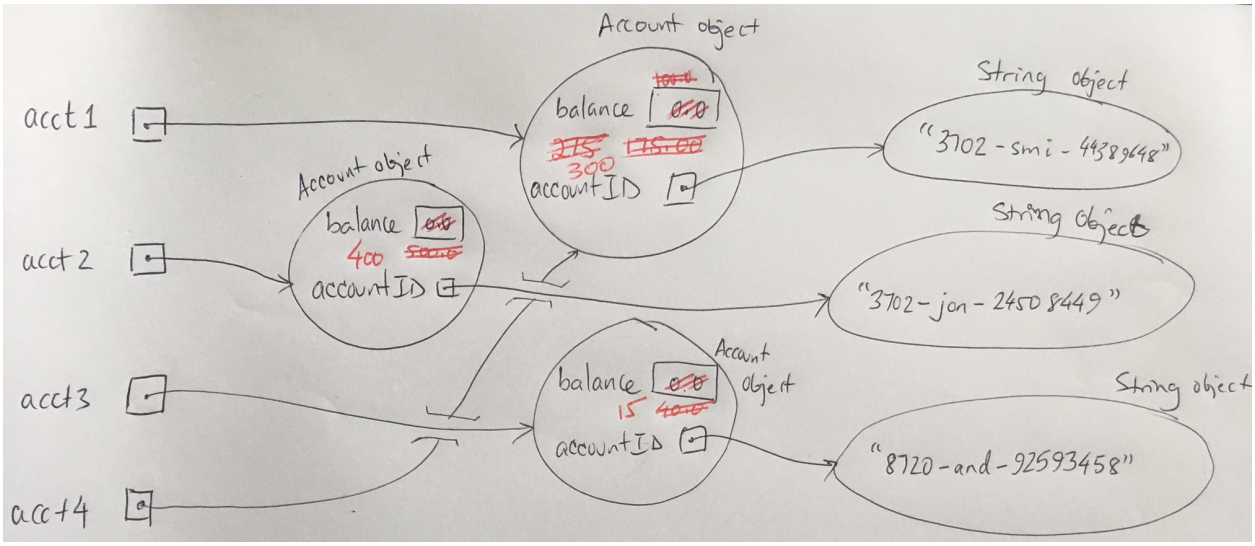
```
transfer(acct2, acct4, 100.0);
transfer(acct3, acct1, 25.0);
transfer(acct1, acct4, 47.0);
transfer(acct1, acct3, 250.0);
```

Here are my diagrams.

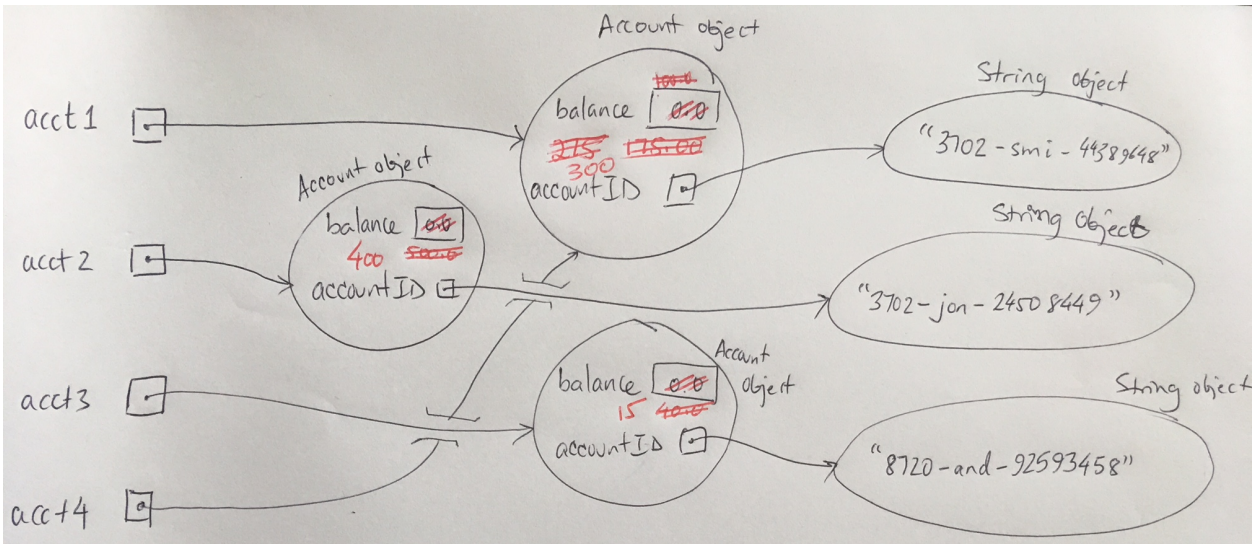
1) after `transfer(acct2, acct4, 100.0);`



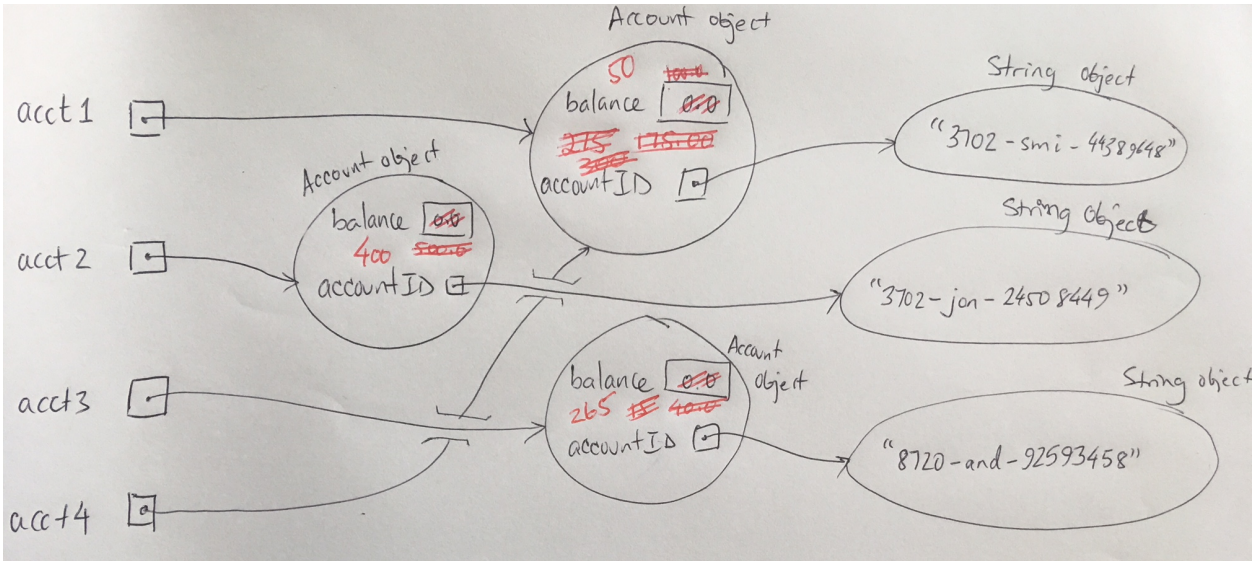
2) after `transfer(acct3, acct1, 25.0);`



3) after `transfer(acct1, acct4, 47.0);` Note: nothing changes: we are transferring from one account to itself!



4) after `transfer(acct1, acct3, 250.0);`



Assessment A:

Consider the following code:

```
Account acct1 = new Account("3702-smi-44389648");
Account acct2 = new Account("3702-jon-24508449");
Account acct3 = new Account("8720-and-92593458");
Account acct4 = acct1;
```

Answer the following questions in the quiz 7a form:

Q1: What is stored in acct1?

A reference to an Account object.

Q2: What is stored in acct4?

A reference to the same Account object as is stored in acct1.

Q3: Can the balance of acct1 differ from the balance of acct4?

No.

Q4: Explain why the balance acct1 can or cannot differ from that of acct4.

acct1 and acct4 both refer to the same Account object.

Engage B, activity1:
Recall the definition of the deposit and withdraw methods of the Account class:

```
public boolean deposit(double amount) {
    if (amount > 0.0) {
        balance = balance + amount;
        return true;
    }
    return false;
}
```

```
public boolean withdraw(double amount) {
    if (amount > 0.0 && amount <= balance) {
        balance = balance - amount;
        return true;
    }
    return false;
}
```

STATE what is printed and EXPLAIN what happens in each of the following method calls:

```
Account a = new Account("A");
Account b = new Account("B");
Account c = b;
System.out.println(a.deposit(100.0));
```

100.0 is deposited into the account named "A".
Because the deposit is successful 'true' is printed.

```
System.out.println(b.deposit(500.0));
```

500.0 is deposited into the account named "B".
Because the deposit is successful 'true' is printed.

```
System.out.println(c.deposit(-100.0));
```

The deposit fails because -100.0 is less than zero.
'false' is printed.

```
System.out.println(a.withdraw(60.0));
```

60.0 is withdrawn from the account named "A".
Because the withdrawal is successful 'true' is printed.

```
System.out.println(b.withdraw(300.0));
```

300.0 is withdrawn from the account named "B".
Because the withdrawal is successful 'true' is printed.

```
System.out.println(c.withdraw(250.0));
```

The withdrawal fails because 250.0 is greater than the current balance of the account named "B".
'false' is printed.

Engage B, activity 2:
Consider these definitions of the deposit and withdraw methods of the Account class:

```
public boolean deposit(double amount) {
    if (amount > 0.0) {
        balance = balance + amount;
        return true;
    }
    return false;
}
```

```
public boolean withdraw(double amount) {
    if (amount > 0.0 && amount <= balance) {
        balance = balance - amount;
        balance = balance - 1.00; // fee
        return true;
    }
    return false;
}
```

STATE what is printed and EXPLAIN what happens in each of the following method calls:

```
Account a = new Account("A");
Account b = new Account("B");
Account c = b;
System.out.println(a.deposit(400.0));
```

400.0 is deposited into the account named "A".
Because the deposit is successful 'true' is printed.

```
System.out.println(b.deposit(500.0));
```

500.0 is deposited into the account named "B".
Because the deposit is successful 'true' is printed.

```
System.out.println(c.deposit(50.0));
```

50.0 is deposited into the account named "B".
Because the deposit is successful 'true' is printed.

```
System.out.println(a.withdraw(199.0) + ": $" + a.getBalance());
```

199.0 is withdrawn from the account named "A", as is the 1.00 fee.
true: $200.00 is printed.

```
System.out.println(b.withdraw(400.0) + ": $" + b.getBalance());
```

400.0 is withdrawn from the account named "B", as is the 1.00 fee.
true: $149.00 is printed.

```
System.out.println(c.withdraw(350.0) + ": $" + c.getBalance());
```

350.0 is greater than the current balance of the account named "B".
false: $149.00 is printed.

```
System.out.println(a.withdraw(200.0) + ": $" + a.getBalance());
```

200.0 is withdrawn from the account named "A", as is the 1.00 fee.
true: $ -1.00 is printed.

Engage B, activity 3:
Consider again the definitions of deposit and withdraw:

```
public boolean deposit(double amount) {
    if (amount > 0.0) {
        balance = balance + amount;
        return true;
    }
    return false;
}
```

```
public boolean withdraw(double amount) {
    if (amount > 0.0 && amount <= balance) {
        balance = balance - amount;
        balance = balance - 1.00; // fee
        return true;
    }
    return false;
}
```

Discuss how the withdraw method can be redefined to prevent an overdraw (ending up with a negative account balance).

The basic idea here is that the problem with the withdraw method is that while it checks that the amount withdrawn is less than or equal to the balance in the account, it does not take into account (no pun intended) the $1.00 fee.

We could re-write the method as follows:

```
public boolean withdraw(double amount) {
    int fee = 1.00;
    if (amount > 0.0 && ( amount + fee ) <= balance) {
        balance = balance - ( amount + fee );
        return true;
    }
    return false;
}
```

Assessment B:
The Account class was originally defined to guarantee that an account's balance could never become negative. More formally, the fact that balance >= 0.0 was guaranteed is called an invariant property.

The fee-charging withdrawal method definition violated this invariant property by allowing the balance to become negative under certain conditions. Here's the relevant part of the class definition:

```
public class Account {
    private double balance;
    private String accountID;
    public Account(String id) { balance = 0.0; accountID = id; }
    /* other definitions omitted */
    public boolean withdraw(double amount) {
        if (amount > 0.0 && amount <= balance) {
            balance = balance - amount;
            balance = balance - 1.00; // fee
            return true;
        }
        return false;
    }
}
```

Answer the following questions in the quiz 7b form:

In the answers the blue text is extra explanation, not necessary for full credit.

Q1: How does the declaration of the balance instance variable help maintain the invariant?

It is private which means it can only be accessed by code inside the Account class body.

Q2: How does the definition of the constructor help maintain the invariant?

It initializes the balance to zero, a value that satisfies the invariant property.

Q3: How does the above definition of the withdraw method allow the invariant to be violated?

It does not account for the fee in the total amount withdrawn.

Q4: How can you "repair" the definition of the withdraw method so in maintains the invariant?

Take the fee into consideration when checking if there are sufficient funds in the account to complete the withdrawal without the balance going negative.