## Exploring Encapsulation, Version 2

**Note –** This exercise is part of the ongoing `Banking` project. You must make sure that you have completed all prior exercises related to this project before attempting this exercise. If you encountered difficulty in completing the prior exercise related to this project, you may obtain the prior exercise's solution files from the `projects\StarterFiles` directory.

In this exercise, you explore the purpose of proper *object encapsulation*. You will modify the `Account` class to hide its data member and provide public methods to manipulate the balance. You will then use the test program that you created in Module 1 to test that the business rule (*balance must not fall below zero*) is satisfied.

Figure 2-2 shows the UML class diagram of the `Account` class that you will create. This design for the `Account` class hides the instance variable, `balance`, and supplies public methods to manipulate the account balance. The `deposit` method adds money to the account. The `withdraw` method removes money from the account. The `getBalance` method returns the current value of the `balance` instance variable.

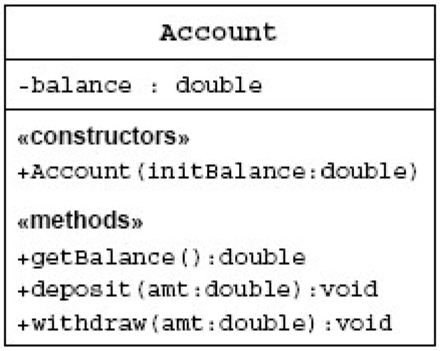
Account
-balance : double
«constructors»
+Account(initBalance:double)
«methods»
+getBalance():double
+deposit(amt:double):void
+withdraw(amt:double):void

**Figure 2-2** UML Class Diagram of Account With Information Hiding

Remember, there is still one business rule that must be maintained: *The balance of the bank account must never go below zero*. Which method must guarantee this business rule?

### Preparation

1. `Go to the projects\BankPrj\` directory.

## Task 1 – Modifying the `Account` Class

Using a text editor, modify the `Account` class source file. This class must satisfy the UML diagram in Figure 2-2.

2. Change the `balance` instance variable from `public` to `private`.

```
private double balance;
```

3. Add the `deposit` method that takes an amount (of type `double`) and adds that amount to the balance. Save the new balance in the instance variable.

```
public void deposit(double amt) {
  balance = balance + amt;
}
```

4. Add the `withdraw` method that takes an amount (of type `double`) and subtracts that amount from the balance. Save the new balance in the instance variable.

```
public void withdraw(double amt) {
  if ( amt <= balance ) {
    balance = balance - amt;
  }
}
```

5. Add the `getBalance` method to return the `balance` instance variable.

```
public double getBalance() {
  return balance;
}
```

## Task 2 – Modifying the `TestAccount` Class

Using a text editor, modify the `TestAccount` program source file. Modify this program to deposit 47 and to withdraw 150.

6. Change the amount in the call to the `deposit` method to `47.0`.

```
acct.deposit(47.0);
```

7. Change the amount in the call to the `withdraw` method to `150.0`.

```
acct.withdraw(150.0);
```

## Task 3 – Compiling the `TestAccount` Program

8. On the command line, use the `javac` command to compile the test program and the `Account` class.

```
javac TestAccount.java
```

## Task 4 – Running the `TestAccount` Program

9. On the command line, use the `java` command to run the test program.

```
java TestAccount
```

The output should be:

```
The final balance is 147.0
```

Notice that the 150 withdraw command did not take affect, because it would have made the balance drop below zero. However, the `Account` object did not tell program that the withdraw command failed, it ignored the command. You will fix this problem in future exercises.