12. Object Oriented Example: a Bank

In this chapter, we're going to look at an example based on a *Bank*. It's going to be a very simple bank – it will basically just consist of *a list of accounts*. Before we get started on the code we need to make a few decisions.

- **What does our bank consist of?** In this case it's pretty easy – there's the *bank* itself and the *accounts* held by customers. Remember in the last chapter that classes (and objects) represent *things*. There are two kinds of thing in this example: the bank, and bank accounts – so we need two classes: `Bank` and `BankAccount`. Actually, in general, we might need more – we might decide that we need extra ones to make the implementation as clear and straightforward as possible. In this (simple) example though, we don't need to do that.
- **What data are we going to store?** For the bank it's easy – we're going to store *accounts*. For accounts we could store quite a lot of information but we're going to restrict ourselves to *names* and *balances*.
- **What things do we want to do with our bank?** Deciding this will decide what *operations* we want, and hence what methods we need. Because we have two classes, we need to separate these into those that belong to an account, and those that belong to the bank. It's easiest if we do that by thinking about account operations and bank operations separately.

Account Operations

We're going to create a very simple bank account – accounts will have a name and a balance. We're going to assume that once an account has been created, the name won't change. This is simple but isn't realistic of course (but neither is using a name to identify an account – you would actually have a unique identifier, usually an account number). This means we're going to need a *constructor* that at the very least has a *name* as a parameter. We can also think about whether it should accept an *opening balance* or not (or if we need two different constructors – one which opens an account with just a name and a balance of £0; and one which also accepts an opening balance). In addition, we are going to need at least three methods as well – one to *check the balance*; one to *deposit* money; and one to *withdraw* money. We might also consider a `toString()` method. The withdraw method raises a question – are we going to allow accounts to go overdrawn? In this case, it's simpler to just say no – accounts cannot have a balance below zero.

Bank Operations

It's not so obvious what operations we would need for the bank itself – partly because there are many possibilities. But we can make a list of some simple basic things we would need to do. Obviously we need to be able to *add*

accounts. It also makes sense to be able to *delete* them too – but what happens to any money in an account when we do this? For now, it's easiest to only allow accounts to be deleted if the balance is zero. We are also going to add two more operations – one to return the *total amount of money* in all accounts; and one to *find* an account if we know the account name.

The Bank Account Class

First we are going to write the account class:

```
public class BankAccount {

     // Account balance – only visible inside the class
     private int balance;

     // Similarly, for the account name
     private final String accountName;

     // constructor with an opening balance
     public BankAccount(String name, int openingAmount){
          balance = openingAmount;
          accountName = name;
     }

     // one with no opening balance - defaults to zero.
     public BankAccount(String name){
          accountName = name;
          balance = 0;
     }

     //Add money
     public int depositMoney(int amount) {
          balance += amount;
          return balance;
     }

     //Withdraw money
     public int withdrawMoney(int amount) {
          if(balance >= amount){
               balance -= amount;
          }
          return balance;
     }
```

```
	//Check the balance
	public int getBalance(){
		return balance;
	}

	//Return name
	public String getName(){
		return accountName;
	}

	//Print out an account:
	public String toString() {
		return("Account Name "+ accountName +
		" has a balance of "+balance);
	}
}
```

We have included two constructors – one which simply sets the name and an opening balance of zero; and the other which sets a specified opening balance. We've also added an extra method that returns the account name – since names are stored in private variables this is the only way to access it. The methods to deposit and withdraw money also return the new balance – we can choose to ignore the value returned if we wish.

Alternative Withdraw Method

One thing to notice is that the method to withdraw money does nothing if there is not enough in the account. An alternative would be to withdraw the entire balance and, instead of returning the new balance, to return the amount actually withdrawn. Here's the alternative code.

```
	public int withdrawMoney(int amount) {
		int amountWithdrawn;
		if(balance >= amount){
			amountWithdrawn = amount;
			balance -= amount;
		} else {
			amountWithdrawn = balance;
			balance = 0;
		}
		return amountWithdrawn;
	}
```

KEY POINT: Be Careful Deciding What Methods Do

Once you've taken a decision like this, you might well be stuck with it for a long time – because others may be using your code. It's easy to change *how things work* but NOT *what they do.* So think about it carefully. There is no reason though why you could not have both withdraw methods in this case – though because both have the same arguments/results they would need different names.

Testing the BankAccount Class

Once you're written a class, write some code to test it – you can make sure it's correct *before* you do the next part of your application. It's *much* easier to test in stages. Here's a test class.

```
import java.util.Scanner;
import java.util.ArrayList;

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

          //Create a single bank account called b
          //and do things to it:
          //deposit and withdraw money
          BankAccount b = new BankAccount("Jones");
          int value = b.depositMoney(500);
          System.out.println(b.getName());
          //two ways of printing balance:
          System.out.println(value);
          System.out.println(b.getBalance());
          System.out.println(b);

          value = b.withdrawMoney(700);
          System.out.println(b.getBalance());

          //Do the same with another account called nb
          BankAccount nb = new BankAccount("Smith",1000);
          value = nb.depositMoney(500);
          System.out.println(nb.getName());
          System.out.println(nb.getBalance());
          //Ignore value returned****
          nb.withdrawMoney(700);
          System.out.println(nb.getBalance());
          System.out.println(nb);
     }
}
```

This isn't great testing – it's not very systematic – and you will do more on testing and how to do it properly in CS-135. Also notice the line with the comment `****` which shows how we can simply ignore the value returned by a method if we want. The method `withdrawMoney()` returns an `int` but we are not assigning it to anything and the compiler is quite happy. We've actually already seen this before - to read a line of text we've written things like:

```
String value = in.nextLine();
```

but we've also written:

```
in.nextLine();
```

to simply ‘throw away’ input (usually after reading an integer).

The Bank Class

Here’s the bank class – mostly it’s straightforward:

```
import java.util.ArrayList;

public class Bank {
     //Store accounts in an ArrayList
     private ArrayList<BankAccount> accountList =
          new ArrayList<>();

     //method to add an account
     public void addAccount(BankAccount account) {
          accountList.add(account);
     }

     //method to delete an account
     public boolean deleteAccount(BankAccount account){
          //only delete 'empty' accounts
          if (account.getBalance() == 0) {
               accountList.remove(account);
               return true;
          } else {
               return false;
          }
     }

     //method to find an account by name
     //variations of this are discussed below
     public BankAccount getAccount(String name) {
          BankAccount findAccount = null;
          int count = 0;
          boolean found = false;
          while((count < accountList.size()) && !found) {
               if (accountList.get(count).
                         getName().equals(name)){
                    found = true;
                    findAccount = accountList.get(count);
               } else {
                    count++;
               }
          }
          return findAccount;
     }
```

```
    //method to return the total amount of money
    public int totalAmount() {
        int total = 0;
        for(BankAccount account : accountList) {
            total += account.getBalance();
        }
        return total;
    }

    //toString method
    public String toString() {
        String accountString = "";
        for(BankAccount account : accountList) {
            //account in code below short for
            //account.toString();

            accountString += account + "\n";
        }
        return accountString;
    }

}
```

We're just going to briefly say something about all these before showing some test code and then talking about `getAccount()` in more detail.

- `addAccount()` – accounts are stored in an `ArrayList` and this method just calls the add() method for the `ArrayList`.
- `deleteAccount()` – again just calls the underlying method in the `ArrayList`, but only if the account has no money in it.
- `getAccount()` – we're going to talk about this one below.
- `totalAmount()` and `toString()` – these are very similar and use a for-each loop to iterate though the list of accounts and, respectively, add the balances; up and call all the `toString()` methods of the individual classes and join the results.

Test Code

Here's the core part of the test code first.

```
Bank testBank = new Bank();

BankAccount testAccount = new BankAccount("Bob", 100);
testBank.addAccount(testAccount);
testBank.addAccount(new BankAccount("Mary"));
testBank.getAccount("Mary").depositMoney(500);
System.out.println(testBank);
```

The 'interesting' lines are:

```
testBank.addAccount(new BankAccount("Mary"));
```

which skips separately creating an account by calling the BankAccount constructor directly in the argument to the addAccount() method for the Bank, and:

```
testBank.getAccount("Mary").depositMoney(500);
```

which:

- Looks up Mary's account (`testBank.getAccount("Mary")`) which returns an object of the `BankAccount` class;
- So we can then directly call the `depositMoney()` method.
- This is equivalent to this:

```
BankAccount tempAccount = testBank.getAccount("Mary");
tempAccount.depositMoney(500);
```

but shorter.

Alternative Way to Create and Add an Account

In the `Bank` and `BankAccount` example, the way we create an account and add it to a bank was is this:

```
String name = "lemon"; //or maybe in.nextLine();
BankAccount newAccount = new BankAccount(name);
bank.addAccount(newAccount);
```

assuming we have created a `Bank` called `bank`. The code inside `Bank` for `addAccount()` looks like this:

```
public void addAccount(BankAccount account) {
     accountList.add(account);
}
```

But we could do it like this instead:

```
String name = "lemon";
bank.addAccount(name);
```

where the code for `addAccount()` is now:

```
public void addAccount(string accountName) {
     accountList.add(new BankAccount(accountName));
}
```

That is, instead of creating the account from the name outside the `Bank`, and passing the new account as a parameter to `addAccount()`, we have just passed the new account name as a parameter to `addAccount()` and we are creating the account inside the `Bank`.

Both approaches are fine in principle – but I am telling you this for a reason…

"Rules are Made to Broken" – getAccount()

Here's the code for getAccount() again:

```
public BankAccount getAccount(String name) {
     BankAccount findAccount = null;
     int count = 0;
     boolean found = false;
     while((count < accountList.size()) && !found) {
          if (accountList.get(count).
                    getName().equals(name)){
               found = true;
               findAccount = accountList.get(count);
          } else {
               count++;
          }
     }
     return findAccount;
}
```

which is pretty long. The reason it's long is because we can't use the 'direct' get method for an `ArrayList` because we're not searching for a particular account, but a name that identifies an account. So we have to loop through the whole list until we find it. And, because we have a 'rule' that says that if we don't know how many times we are going to iterate around a loop we need to use a while loop. However, if we 'break' that rule – and the one we have about exiting loops mid-way through them we can rewrite this:

```
public BankAccount getAccount(String name) {
          for (int i = 0; i < accountList.size(); i++) {
              if(accountList.get(i).getName().
                  equals(name)){
                    return accountList.get(i);
               }
          }
          return null; //if not found
     }
```

which is quite a bit shorter, and, probably, a bit clearer. This is very similar to the 'exception' we talked about in the Loops chapter, allowing `break` in `for` loops – we are doing the same here, except we are using return instead. The difference is that we are now returning a value from the middle of the for loop, and so ending it early – meaning it doesn't loop as many times as we expected at the start. With experience you'll come to 'break' some of the rules

we've introduced – if you didn't read it at the time, go back and look at the last section of the Loops chapter. Here's an even better way:

```
public BankAccount getAccount(String name){
     for(BankAccount account: accountList){
          if(account.getName().equals(name) {
               return account;
          }
     }
     return null;
}
```

Advanced Aside: An Even Better (but Tricky) Way..

***Absolutely* skip this if new to programming.** An even shorter way to do this is to use the new (to Java) *functional*, or *lambda*, notation:

```
public BankAccount getAccount(String name){
     BankAccount account =
          accountList.stream().filter(a -> a.getName().
          equals(name)).findFirst().get();
     return account;
 }
```

I'm not going to explain this in the notes because it depends on understanding material that you won't see until the second year in the *Concurrency* and *Declarative Programming* modules. But as well as being shorter, code like this is easier to *automatically parallelize* – that is, execute in parallel on processors with multiple cores (which most computers have these days).

Using Bank 'Properly'

The test code above is a bit basic and doesn't really show much about how we can use the class – really, we'd like some kind of interface that lets us choose to create and delete accounts; and add and withdraw money from them. We're going to create a `BankMenu` class that provides a basic text-based menu interface letting us do just that.

Advanced Aside

Really, if we were going to do this, we'd use some more advanced concepts in Java to make it more flexible – it would be nice, for example, if the menu class we wrote was *generic* in that we could, using parameters, specify what it says and does (instead of hard-coding that information in the code). We can do that by using things like *reflection* and *interfaces*, but they are beyond the scope of this module

Confession and *Refactoring*

The code for this class is a bit flaky - it works but it was pretty late when I wrote it. It could really do with some tidying-up: broken down a bit more into methods; some renaming; some constants. This is something you typically do

after you've got your code working and it falls under the heading of *refactoring* – things you do to make the presentation and structure of the code better but without changing how it works. (I wrote that in 2015 and I could, of course, have re-written the code since them: but I've left it in to illustrate the *concept* of refactoring – tidying up your code.)

The RunBank Class

The first class we're going to write is the one containing the main method. It's pretty simple:

```
public class RunBank {
     public static void main(String[] args) {
          Bank bank = new Bank();
          BankMenu menu = new BankMenu(bank);

          while(menu.display());
     }
}
```

The main method:

- First creates an empty bank:
  `Bank bank = new Bank();`
- And then creates a `BankMenu` object, passing the new bank as a parameter:
  `BankMenu menu = new BankMenu(bank);`

We haven't written `BankMenu` yet – we'll do that next. However notice that would could have simply decided to create the empty bank inside the `BankMenu` constructor. *But remember we said that methods should do one thing* – the same is true of classes. In this case, the `BankMenu` class is only going to handle the interaction with the bank – not create it. Suppose we wanted it to work with a bank that already existed and wasn't initially empty (maybe read in from a file). If we created an empty bank inside BankMenu, we couldn't do that.
The last line might be confusing:

```
while(menu.display());
```

This simply says that we are going to repeatedly call a method belonging to the `BankMenu` object called `display()` until it returns false. Because all the work is actually going to be done inside `display()`, the body of the loop is actually empty. But we could rewrite this:

```
boolean cont = true;

do {
     cont = menu.display();
while(cont);
```

This does the same thing, and maybe a bit clearer initially – but the first way is very short, and a common way to approach problems like this (if it wasn't common and therefore widely understood by programmers, I wouldn't recommend doing things like this). Now we need to write the much longer `BankMenu` class.

BankMenu

The `BankMenu` class is quite long and we're not going to show the whole thing in one go. Instead, we're going to do it piece by piece. First the variables (fields) and the constructor

```
private Bank bankDetails;
private Scanner in;

public BankMenu(Bank bank) {
     bankDetails = bank;
     in = new Scanner(System.in);
}
```

The constructor simply stores the bank passed in as a parameter within a private variable called `bankDetails`, and creates a new `Scanner` to read in data.

The display() Method

There is only one public method in `BankMenu` – and it's pretty long. We saw it called in `RunBank` above: the `display()` method.

```
public boolean display() {
      boolean cont = true;
      BankAccount account;

      System.out.println("Bank Options");
      System.out.println(
              "Select option from list: \n" +
              "  1 - Add an account\n" +
              "  2 - Delete an account\n" +
              "  3 - Deposit money\n" +
              "  4 - Withdraw money\n" +
              "  5 - Print Details\n" +
              "  0 - Exit\n"
       );

       int selection = readInt("");
       in.nextLine();

       switch (selection) {
       case 1: //Add account
            account = newAccount();
            String newName = account.getName();
            if ((account != null) && (bankDetails.getAccount(newName)
                        == null)){
                  bankDetails.addAccount(account);
                  System.out.println("Added OK");
            } else {
                  System.out.println("Name not unique or cancelled");
            }
            break;
      case 2: //Delete account
            account = readAccount();
            if ((account != null) &&
                        bankDetails.deleteAccount(account)){
                  System.out.println("Deleted OK");
            } else {
                  System.out.println("Not empty or cancelled");
            }
            break;
      case 3: //Deposit Money
            account = readAccount();
            if ((account != null)) {
                  int amount = readInt("Enter amount: ");
                  account.depositMoney(amount);
                  System.out.println("Balance: " +
                        account.getBalance());
            } else {
                  System.out.println("No changes");
            }
            break; //Switch continues on next page
```

```
        case 4: //Withdraw Money
              account = readAccount();
              if ((account != null)) {
                    int amount = readInt("Enter amount: “);
                    int withdrawn = account.withdrawMoney(amount);
                    System.out.println("Withdrawn: " + withdrawn);
                    System.out.println("Balance: " +
                          account.getBalance());
              } else {
                    System.out.println("No changes");
              }
              break;
        case 5: //Print details
              System.out.println(bankDetails);
              break;
        case 0: //Quit
              cont = false;
              break;
        default: //Error
              System.out.println("Invalid selection.");
              break;
        }
        return cont;
}
```

This method:

- Prints out a simple menu.
- Reads in the user choice using a method `readInt()` defined later (the parameter is the prompt that will be printed and we don't need it here).
- Uses a large switch statement to select between options.
- Option 1 is to add an account – we create one using the `newAccount()` method defined below. Provided the account is not `null` (which means the user *cancelled* the operation) and the name does not already exist, it's added and an appropriate message printed.
- Option 2 is similar and deletes an account – the method `readAccount()` defined below will either return an account in the bank or null if it wasn't found or the user *cancels*. Provided the account is not null and is empty, it's deleted.
- Option 3 deposits money – once again we use `readAccount()` to get an account and once again null means the user *cancelled* – provided they did not, the amount to be deposited is read in and deposited.
- Option 4 withdraws money – it's very similar to depositing except that the amount withdrawn depends on the available balance.
- Option 5 just prints the current account names and balances
- Option 0 sets the return value of the method to `false` meaning that the loop that calls it in `RunBank` will end.

Give Users an Escape Route

Notice that in the text above we several times said 'if the user cancelled'. In general, it's *always* good practice to give users the chance to back out of operations – we could easily have not allowed them to do that in this case.

But what if they accidentally selected the deposit option with no way to cancel: they have to enter a valid account name (what if they don't know one) to get any further and, if in that case they have no option but to exit the program.

The Other Methods

There are four other – private – methods in `BankMenu` – they are below with just a brief statement about what they do.

The first method just reads an integer – it turns any negative numbers into positive ones, which isn't great but makes the code shorter and simpler which is OK here.

```
private int readInt(String prompt) {
     System.out.print(prompt);
     while(!in.hasNextInt()){
          System.out.print("Enter an integer: ");
          in.nextLine();
     }
     int value = in.nextInt();
     return Math.abs(value);
}
```

Next is a method to read in account names – it's very simple:

```
private String enterAccountname() {
     System.out.println("Enter an account name");
     return in.nextLine();
}
```

Next is the method to create a new account – the only complication is that if the user does not enter a name we assume they are cancelling the operation. This method doesn't check if the name entered is unique – that's done in the display() method above.

```
private BankAccount newAccount() {
     System.out.print("Enter a new account " +
          "name or just enter to cancel: ");
     BankAccount newAccount = null;
     String newName = in.nextLine();
     if (!newName.equals("")) {
          newAccount = new BankAccount(newName);
     }
     System.out.println(newAccount.getName());
     return newAccount;
}
```

Finally, here's the method that gets an account corresponding to a name. This keeps looping and asking until the user either enters a name of an actual account or cancels by just typing return.

```
private BankAccount readAccount(){
     BankAccount account = null;
     boolean done = false;
     do {
          System.out.print("Enter the name of “+
               “an existing account or “+
               “enter to cancel: ");
          String accountName = in.nextLine();
          if (accountName.equals("")) {
               done = true;
          } else {
               BankAccount provAccount =
                bankDetails.getAccount(accountName);
               if (provAccount!= null){
                    done = true;
                    account = provAccount;
               }
          }
     } while(!done);
     return account;
}
```