Department of Computer Science

08120 Programming 2
Week 31 Practical 2013/2014
Bank Test Data

This week we are going to work with a partially completed bank system. You will have to create some test data and also add some features to the bank accounts.

The Friendly Bank Account Management System

We have got a partially completed bank management system to work with. It contains classes that implement the `Account` and `Bank` behaviours. You can find the partially completed code in a project called Bank which is stored on the module page for this site:

http://intra.net.dcs.hull.ac.uk/student/modules/08120/default.aspx

You should download the starter project and unzip it. The program will run, create a single account, add it to the bank, save the bank in a file and then reload the bank. Note that the classes implement the data storage for the objects in the bank, but a lot of functionality is missing. However, for the purpose of this practical work you have all the behaviours you need.

Adding ToString methods

Before you can work on the software you need to download and build it.

Before you go any further; perform the following:

1. Make a folder called lab31 in your filestore and download the **`Bank.zip`** file from the laboratory web page on SharePoint.
2. You now need to extract the Bank project from the archive and copy it into your folder.
3. Open the Bank solution with Visual Studio.
4. Compile and run the program. It doesn't print anything on the screen but it should run successfully.

At the moment the `Account` and the `Bank` class do not have `ToString` methods. The customer has asked for the following:

- The `ToString` method for the `Account` should return the name, address, balance and account number for the account, as a string
- The `ToString` method for the Bank should return the name of the bank and the number of accounts in the bank, as a string.

Before you go any further; perform the following:

5. Add the `ToString` methods to those two classes. Remember that the `ToString` method does **not** write anything to the user, instead it just returns a string that describes the contents of the object.

Adding an Equals method for the Account class

Once you have working `ToString` behaviours the next thing to add is an `Equals` behaviour for the `Account` class. This will return `true` if the object that it is being compared with contains the same data.

```
public override bool Equals(object obj)
{
    return true;
}
```

Above is an "empty" equals behaviour that always returns true, i.e. it says that all the objects are always equal. You need to make this method return `false` if any of the items in one `Account` object (the name, address, balance, account number etc) are not the same as those in the other.

Before you go any further; perform the following:

6. Add an Equals behaviour to the Account class. You can find out more about equals behaviours in the "Object Etiquette" slide deck for this week.

You can test your Equals methods with the following code.

```
Account a = friendlyBank.AddAccount("Rob", "Hull", 100);
Account b = friendlyBank.AddAccount("Rob", "Hull", 100);

if (a.Equals(b))
{
    Console.WriteLine("Test passed");
}
else
{
    Console.WriteLine("Test failed");
}
```

Before you go any further; perform the following:

7. Test your equals behaviour using the above code.

Of course the test above only checks one behaviour. If the method always returns true (as it did originally) this would pass the above tests. When you test code you need to test every possibility

Before you go any further; perform the following:

8. Expand the test above so that the equals behaviour is tested for every possible option. This might mean creating several tests, which is fine.

Adding an Equals method for the Bank class

It would be very useful to be able to compare two banks to make sure that they hold the same data. This would be how you would test the save and load behaviour. The Equals behaviour for the Bank class must use the Equals behaviour from the Accounts in it. The comparison sequence could be as follows:

- If the banks have different names they are different
- If the banks have different numbers of accounts they are different

- For each account in one bank, find the corresponding account in the other bank and if they are different, the banks are different

This is quite a complicated behaviour, but it is necessary, otherwise you will never know if your save and load behaviours for the bank work correctly.

Before you go any further; perform the following:

9. Add an Equals behaviour for the bank.

Creating Test Data

If you want to create a convincing set of tests for your bank you need to have more than just one or two accounts. It turns out this is very easy to do.

Creating Test Names

Each test account should really have a name that makes sense. Fortunately we can create a large number of names using a program.

```
string[] firstNames = new string[] { "Rob", "Fred", "Jim",
    "Ethel", "Nigel", "Simon", "Gloria", "Evadne" };
string[] surnames = new string[] { "Bloggs", "Smith",
        "Jones", "Thompson", "Wooster", "Brown", "Acaster",
        "Berry", "Ackerman" };

foreach (string surname in surnames)
{
    foreach (string firstname in firstNames)
    {
        Console.WriteLine(firstname + " " + surname);
    }
}
```

The C# above creates two arrays of strings, one contains surnames and the other firstnames. The for loops then work through the lists generating every possible name permutation. At the moment the names are just written to the screen, but you can sue them to create bank accounts. You can create test addresses in a similar way, but making lists of towns and picking random ones from it.

Creating Test Balance values

Once you have created the names you now need some balance values to test with. The best way to do this is by creating random numbers;

```
Random r = new Random(1);

Console.WriteLine(r.Next(-100,10000));
```

The above code creates a random number generator called r. It is created with the seed value 1, which means that it will always produce the same sequence. The code then asks for a number in the range -100 to 10000 that we can use for the initial balance of our accounts. This will always print out 2411 as this is the first number in the sequence produced from a seed of 1. Note that if you ask for a random number from a different range (i.e. not -100 to 10000) then you will get a different value.

Getting the same sequence each time is useful because it means that you can make sure that each time you create the test bank it has exactly the same set of balance values.

Before you go any further; perform the following:

10. Create a test bank that contains 80 customers.
11. Save the bank into a file and then load it back. Use the equals behaviour for the bank to make sure it has been saved correctly.

Adding an Overdraft value to Accounts

The Bank Manager would like you to add an extra value to each account stored in the bank. The value is an overdraft value, which is set for each account and allows the account holder to go "overdrawn" on their account. In other words, if the overdraft is 100 pounds this means that the customer balance can fall to -100 (i.e. 100 pounds overdrawn).

The overdraft for an account is initially set to 0. The `Account` class will need to provide a `GetOverdraft` and a `SetOverdraft` method to manage the overdraft.

Before you go any further; perform the following:

12. Add overdraft storage to the system. You will have to modify the Account save, load, ToString and Equals methods to manage the new account data. You should **not** have to make any changes to the Bank class. Make sure that you test the behaviour of the account with the new data.

Rob Miles
February 2014