UNIVERSITY OF Hull

Saving and Loading

Rob Miles

Department of Computer Science

Data Storage

- At the moment the data in the bank is lost when the bank program stops running
- You could not run an actual bank program like this
- We need a way that the program can use files to hold bank account details

Bank Program Use

- Use of the Bank Account program will work as follows
    1) Start the program
    2) Load the bank information
    3) Work on the accounts
    4) Save the bank information
    5) Stop the program

Saving the Bank Information

- We already know how to write data into a file
  - The program must create a stream and use this to write to the file
- This is how we are going to save the bank information
- We need to save the `Account` and the `Bank` information to the file

Who does what?

- The only thing that can save and load account information is the `Account` class
- The only thing that can save and load bank information is the `Bank` class
- We need to put the save and load behaviours into these classes

Account Save Method

```
public bool Save(System.IO.TextWriter textOut)
{
    try
    {
        textOut.WriteLine(accountNumber);
        textOut.WriteLine(name);
        textOut.WriteLine(address);
        textOut.WriteLine(balance);
    }
    catch
    {
        return false;
    }
    return true;
}
```

Saving to a Stream

- The `Save` method you have just seen is given a reference to the `TextWriter` stream to be used to save the data:

```
public bool Save(System.IO.TextWriter textOut)
```

- We actually want to save to a file
  - This method saves to a stream
- Does this mean we have made a mistake?

Using the stream Save method from another Save method

- We can use this version of the Save to save to a file:

```
public bool Save(string filename)
{
    System.IO.TextWriter textOut =
            new System.IO.StreamWriter(filename);
    if (Save(textOut))
    {
        textOut.Close();
        return true;
    }
    return false;
}
```

Overloading

- We now have two versions of the `Save` method
- One saves to an already opened `Stream`
- The other saves to a file
- This is an example of overloading in action
- It is often appropriate to provide multiple methods to do the same thing
- It is also very sensible to make one overloaded method call another

Account Save Method Error Handling

```
public bool Save(System.IO.TextWriter textOut)
{
    try
    {
        textOut.WriteLine(accountNumber);
        ...
    }
    catch
    {
        return false;
    }
    return true;
}
```

- The `Save` method returns `false` if it doesn't work
- It does this by catching any exceptions

Error Handling

```
if (robsAccount.Save("Data.txt")
{
    Console.WriteLine ("Saved OK");
}
```

- This version of `Save` returns `false` if things go wrong
- Is this the best thing to do?
  - Stops the program from throwing exceptions
  - Might hide things that go wrong – a programmer that uses `Save` has to make sure they test the result returned
  - Makes things much more complicated
  - Makes things harder to test

Bad things about catching Exceptions

- If you catch an exception you are hiding information
  - There was a reason why the exception was thrown, if you catch the exception this reason may be hidden
- Therefore it might be best to leave your caller to pick up the pieces, rather than deal with it yourself
- At least with an exception they will have their attention drawn to the event

A Simpler Save Method

```
public void Save(System.IO.TextWriter textOut)
{
    textOut.WriteLine(accountNumber);
    textOut.WriteLine(name);
    textOut.WriteLine(address);
    textOut.WriteLine(balance);
}
```

- This version of `Save` does not return whether it worked or not
- It just throws exceptions if it fails
- These must be handled by the caller

Exception and Design

- The worst thing that could happen is if someone uses your method and thinks it has worked when it hasn't
- At least an exception being thrown will make it clear that something has gone wrong
- From now on I'm going to take this approach
- Remember that in a real project you would have to set standards for your error handling

Simple File Save

```
public void Save(string filename)
{
    System.IO.TextWriter textOut =
            new System.IO.StreamWriter(filename);
    Save(textOut);
    textOut.Close();
}
```

- This version looks OK, but it has a problem
- It calls the other version of `Save`
- If this fails it might leave `textOut` open on the file

Dangling Streams

- A program should never leave a stream open
- The stream should be closed, even (or perhaps especially) if the write operation fails
- Failure to do this might cause problems later on if the program tries to access the same file
- In this case there is a chance that the garbage collector will fix the problem, but you can't rely on this

Proper File Save

```
public void Save(string filename)
{
    System.IO.TextWriter textOut = null;
    try
    {
        textOut = new System.IO.StreamWriter(filename);
        Save(textOut);
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        if (textOut != null) textOut.Close();
    }
}
```

Catch and Finally

- We have seen `try - catch` constructions before
- Reading from a file and parsing numbers can both throw exceptions which must be caught
- Whether the exception is thrown or not we still have to close the file
- The `finally` part of a `try - catch` construction lets us specify code that is always obeyed, irrespective of whether or not the exception is thrown

The Finally Part

```
try
{
    textOut = new System.IO.StreamWriter(filename);
    Save(textOut);
}
catch (Exception e)
{
    throw e;
}
finally
{
    if (textOut != null) textOut.Close();
}
```

- The `finally` part of the `try - catch` is always obeyed, this makes sure that the file is closed whatever happens to the read operation

Re-throwing Exceptions

- We need to make sure that when something goes wrong, the thing that called us is made aware of this
- Since we are passing exceptions to our caller we must re-throw any which are caught by our method
- This provides our caller with the best chance of finding out what when wrong

Re-Throwing Exceptions

```
try
{
    textOut = new System.IO.StreamWriter(filename);
    Save(textOut);
}
catch (Exception e)
{
    throw e;
}
finally
{
    if (textOut != null) textOut.Close();
}
```

- The `catch` clause can be given the exception that was thrown
- It can then re-throw it

Saving an Account

```
Account rob = friendlyBank.AddAccount("Rob", "Hull", 100);
rob.Save("test.txt");
```

- This code creates a bank account and saves it in a file called `"test.txt"`

```
5
Rob
Hull
100
```

- This is the output from the call of `Save`
- Each item is placed on a separate line
- Rob is account number 5

Loading an Account

- We can now ask an `Account` instance to save itself by calling its `Save` method
- However, we can't ask an `Account` to load itself:
  - At the time of the load we don't have an account to load
- We can solve this by using a `static` method in the `Account` class to load the account for us

Static Load Method

- Because the `Load` method is part of the class, not part of an instance, it is always present
- It can return a reference to the `Account` that it created from the saved data
- If this process fails it will signal this by throwing an exception

Account Load Method

```
public static Account Load(System.IO.TextReader textIn)
{
    int accountNumber = int.Parse(textIn.ReadLine());
    string nameText = textIn.ReadLine();
    string addressText = textIn.ReadLine();
    string balanceText = textIn.ReadLine();
    decimal balanceValue = decimal.Parse(balanceText);
    return new Account( nameText,addressText,
                        balanceValue,accountNumber);
}
```

- The `Load` method reads in all the data items and then uses them to build a new `Account` and return it
- If it fails it will throw an exception

Loading from a File

- To load an `Account` from a file the method must:
    1. Open the file
    2. Read the Account
    3. Close the file
- It is very important that the file is not left open if reading the account fails
- We need to deal with exceptions

Account Load Method

```
public static Account Load(string filename)
{
    Account result;
    System.IO.TextReader textIn = null;
    try
    {
        textIn = new System.IO.StreamReader(filename);
        result = Load(textIn);
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        if (textIn != null) textIn.Close();
    }
    return result;
}
```

Loading an Account

```
Account rob = friendlyBank.AddAccount("Rob", "Hull", 100);
rob.Save("test.txt");
Account robCopy = Account.Load("test.txt");
```

- This code creates a bank account and saves it in a file called "`test.txt`"
- It then loads it back into a different object
- If the load and save works these two objects should contain the same account data

Testing Individual Accounts

- We can test the saving and loading of accounts without needing a working Bank
- The account is a separate object that works independently of the bank
- This is exactly how the system would be designed

Saving a Bank

- We can use exactly the same behaviours to save an instance of the `Bank` class
- The `Bank` will ask each account in the bank to save itself
- We also need to save how many accounts are being saved
- This is so that the load behaviour can read in the correct number of accounts

Saving a Bank to a stream

```
public void Save(System.IO.TextWriter textOut)
{
    textOut.WriteLine(bankName);
    textOut.WriteLine(newAccountNumber);
    textOut.WriteLine(bankAccounts.Count);
    foreach (Account a in bankAccounts)
    {
        a.Save(textOut);
    }
}
```

- This writes the name of the bank, the number of the next new account and how many accounts there are in the bank
- It then writes out each account in turn

Account save

```
public void Save(System.IO.TextWriter textOut)
{
    textOut.WriteLine(bankName);
    textOut.WriteLine(newAccountNumber);
    textOut.WriteLine(bankAccounts.Count);
    foreach (Account a in bankAccounts)
    {
        a.Save(textOut);
    }
}
```

- It is important to note that it is actually the account that saves itself, the bank simply passes the account the stream to use when it is saved

Saving a Bank to a File

```
public void Save(string filename)
{
    System.IO.TextWriter textOut = null;

    try
    {
        textOut = new System.IO.StreamWriter(filename);
        Save(textOut);
    }
    catch (Exception e)
    {
        throw e;
    }
    finally
    {
        if (textOut != null) textOut.Close();
    }
}
```

Notice Anything?

- The method that saves a `Bank` to a file is **exactly** the same as the method that saves an `Account`
- This is just as it should be
  - Posh programmers call these *patterns*
- There are even tricks in C# which will let us reuse code this way

Saving collections of Banks

- If we wanted to create a collection of Banks (perhaps for a large company) we can simply replicate the pattern that we now have
- This illustrates the power of using objects and methods to manage the data and provide behaviours

Summary

- You must add `Load` and `Save` behaviours to your classes
- You should save to streams and files
- The `Load` method must be `static`
- It is best if you leave error handling to the method that calls you
- Make sure that you never leave an open file