


Awarding
Great British
Qualifications




Designing and Developing Object-Oriented Computer Programmes
Topic 11:
File IO

1

Scope and Coverage

This topic will cover:

- Writing to Files
- Reading from Files
- Choosing Files
- Serialization
- Using Serialization




2

Learning Outcomes

By the end of this topic students will be able to:

- Read data from external files;
- Write data to external files;
- Use dialog choosers for files;
- serialization of objects;
- Deserialization of objects;




3

Title of Topic: Topic 1 - 1.4

Introduction - 1

- Up until this point, we have covered a large number of the fundamental building blocks of C#, but we are hugely restricted in that we cannot possibly write 95% of real world applications.
- That's simply because we have no way of storing the state of an application between executions.
- We run our code, we put in some data, we close the application down – and everything is lost.




4

Title of Topic: Topic 1 - 1.5

Introduction - 2

- We might use constructor methods to populate test data, but as soon as the program stops running our data is gone.
- Luckily though the C# programming language provides us with a wide range of methods and objects for dealing with File Input/Output (File IO) requirements.




5

Title of Topic: Topic 1 - 1.6

Introduction - 3

- In this chapter, we'll be looking at one of the simpler frameworks for loading and saving data, that of **stream based IO**.
- Then we'll look at serialization as a way of storing object data indefinitely.



6

Title of Topic: Topic 1 - 1.7

Writing to Files - 1

- Let's look at how we might write a very simple text editor.
- This is going to have a large text area in the centre, and two buttons – 'load' and 'save'.
- We'll allow the user to enter a file path, and then that's where in the file system we'll work with the document.

NCC

7

Title of Topic: Topic 1 - 1.8

Writing to Files - 2

- With stream based IO we have a three phase process that lets us make a connection to a file.
- First, we open a connection to a file location that we specify.
- We do this through the use of a StreamWriter.
- We'll need to add the System.IO **using** definition at the top of our programs to get access to this.
- Our application is going to look like this:

NCC

8

Title of Topic: Topic 1 - 1.9

Writing to Files - 3


NCC

9

Title of Topic: Topic 1 - 1.10

Writing to Files - 4

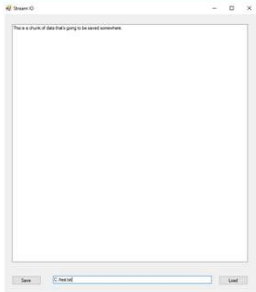
- Our largest text area is named txtInput, the smaller txtLocation.
- The buttons are cmdSave and cmdLoad respectively.
- All we're going to do here when we press the save button is take the text in txtInput and store it in a file indicated by the txtLocation textbox, like so:




10

Title of Topic: Topic 1 - 1.11

Writing to Files - 5





11


Title of Topic: Topic 1 - 1.12

Writing to Files - 6

- Adding the first bit of code for our save button gives us the following:

```
private void cmdSave_Click(object sender, EventArgs e)
{
    StreamWriter output;

    output = new StreamWriter(txtLocation.Text);
}
```




12

Title of Topic: Topic 1 - 1.13

Writing to Files - 7

- The StreamWriter object contains a number of methods that give us access to the functionality we need to write things to the file.
- One of the things it has is a WriteLine function that lets us take a chunk of text and write it directly into the file we specify.
- If the file doesn't exist, it'll be created for us.



13


Title of Topic: Topic 1 - 1.14

Writing to Files - 8

- Once we're done, we **close** the file to complete the interaction and release the file back to other applications to use if they want:

```
private void cmdSave_Click(object sender, EventArgs e)
{
    StreamWriter output;

    output = new StreamWriter(txtLocation.Text);
    output.WriteLine(txtInput.Text);
    output.Close();
}
```




14

Title of Topic: Topic 1 - 1.15

Writing to Files - 9

- For alphanumeric data, this is fine.
- Objects won't work with this system, but we'll see how to fix that a little later in this chapter.



15

Title of Topic: Topic 1 - 1.16

Reading from Files - 1


- Next, we want to be able to load in a file that we previously created.
- As you might imagine, this works in a very similar way except we use a **StreamReader** instead.
- We also make use of the ReadToEnd method to read in the entirety of the data in a single go:

```
private void cmdLoad_Click(object sender, EventArgs e)
{
    StreamReader input;

    input = new StreamReader(txtLocation.Text);

    txtInput.Text = input.ReadToEnd();

    input.Close();
}
```




16

Title of Topic: Topic 1 - 1.17

Reading from Files - 2

- While reading in text input is roughly the same complexity as writing it, we often have to do other things to turn that input into useful data.
- For example, we might write out an array of integers into a string and then have to turn the string that comes back into an array of integers.
- This is a process called **parsing** and it is highly individualised for each program.




17

Title of Topic: Topic 1 - 1.18

Choosing Files - 1

- The problem with this program is that we're currently using a textbox to provide a filename, and that's very fragile and unfriendly to users.
- There are reasons we might want to do it, but usually it's better to give the user an easier option – a file open dialog.
- Let's look at changing our program now so that the txtLocation box goes away and we use a **chooser** instead.



18

Title of Topic: Topic 1 - 1.19

Choosing Files - 2


- Let's look at how it works for saving:

```
private void cmdSave_Click(object sender, EventArgs e)
{
    StreamWriter output;
    SaveFileDialog ofd;
    String file;
    DialogResult selected;

    ofd = new SaveFileDialog();
    selected = ofd.ShowDialog();

    if (selected == DialogResult.OK)
    {
        file = ofd.FileName;
    }
    else
    {
        return;
    }

    output = new StreamWriter(file);
    output.WriteLine(txtInput.Text);
    output.Close();
}
```




19

Title of Topic: Topic 1 - 1.20

Choosing Files - 3

- First, we create the SaveFileDialog and tell it to show itself.
- What comes back from that is a DialogResult object that we use to compare to some fixed results that are held in the DialogResult class itself.
- If the OK button was pressed, we grab the filename that was selected.
- Otherwise, we return from the function.
- The save file dialog will do all the rest of the work of handling confirmation and checking for the existence of the files. It's very handy.



20

Title of Topic: Topic 1 - 1.21

Choosing Files - 4


- Opening a file works in a very similar way, except we use an OpenFileDialog like so:

```
private void cmdLoad_Click(object sender, EventArgs e)
{
    StreamReader input;
    OpenFileDialog ofd;
    String file;
    DialogResult selected;

    ofd = new OpenFileDialog();
    selected = ofd.ShowDialog();

    if (selected == DialogResult.OK)
    {
        file = ofd.FileName;
    }
    else
    {
        return;
    }

    input = new StreamReader(file);
    txtInput.Text = input.ReadToEnd();
    input.Close();
}
```




21

Title of Topic: Topic 1 - 1.22

Choosing Files - 5

- Not only does this give us a lot of functionality for free, it also makes the application work better for the user and behave consistently with all the other applications we might use on a day to day basis.
- It adds a little bit of extra work to every File IO interaction, but makes them much stronger as a result.




22

Title of Topic: Topic 1 - 1.23

Serialization - 1

- Objects are a little more complex than string data – they’re usually made up of a number of other data fields of their own, for one thing.
- The process of **serialization** is used to turn an object into a stream of bytes that can be stored on a file (or a database, or transmitted via a network, or whatever you want to do with it).




23

Title of Topic: Topic 1 - 1.24

Serialization - 2

- This is how we save the state of an object between invocations.
- Let’s go back to our cash machine and see how we could use serialization to save the state of our bank.




24

Title of Topic: Topic 1 - 1.25

Serialization - 3

- We're going to want to be able to **serialize** the classes we have created for our objects – nothing else in this program needs to save.
- When we save we're going to want to take the List of those accounts and write them out to a file.
- When loading, we want to take the serialized stream of bytes and turn it back into a list of objects.




25

Title of Topic: Topic 1 - 1.26

Serialization - 4

- Luckily, it's easy enough to do but we do need to change our classes a little to indicate which of our attributes are serializable.
- If we had an object made up of other objects, we'd need to go all the way down the object chain making sure every single part of it could be converted into a byte.
- If we don't do this, we'll get an exception when we make the attempt.




26

Title of Topic: Topic 1 - 1.27

Serialization - 5

- First, we amend our classes like this:

```
[Serializable()]
class Account : ISerializable
{
```




27

Title of Topic: Topic 1 - 1.28

Serialization - 6

- What we're saying here is that we're going to have a set of methods in our account that implement a special kind of structure called an **Interface**.
- What that means is that we now need to add in two new methods.
- One is a **deserialization constructor** that is used when we load in an object, and the other is `GetObjectData` which is what we use to strip off key elements of data and put them into a key/value pair.




28

Title of Topic: Topic 1 - 1.29

Serialization - 7

- We'd implement the constructor like so:

```
public Account (SerializationInfo info, StreamingContext context) {  
    PIN = (String)info.GetValue ("PIN", typeof(String));  
    accountNumber = (String)info.GetValue ("Account Number", typeof(String));  
    balance = (int)info.GetValue ("Balance", typeof(int));  
    lastTransaction = (List<String>)info.GetValue ("Last Transaction",  
    typeof(List<String>));  
}
```




29

Title of Topic: Topic 1 - 1.30

Serialization - 8

- Note here that there's an unusual syntax, but all you need to know is that for the bracketed part before the `getValue` call, you use the data type of the attribute you want to set.
- For the `typeof` you do exactly the same thing.
- The first string that goes into `GetValue` is something we'll decide for ourselves, as part of the `GetObjectData` call:




30

Title of Topic Topic 1 - 1.31

Serialization - 9

```
public virtual void GetObjectData (SerializationInfo info, StreamingContext context)
{
    info.AddValue ("PIN", PIN);
    info.AddValue ("Account Number", accountNumber);
    info.AddValue ("Balance", balance);
    info.AddValue ("Last Transaction", lastTransaction);
}

```




31

Title of Topic Topic 1 - 1.32

Serialization - 10

- You don't have to worry about calling either of these functions.
- It'll be done for us. But we also need to make sure all our specialised classes are **also** serialized.
- We use the same basic system for this, except we need to call the base constructor (as we have seen before) and override GetObjectData with our new data.



32

Title of Topic Topic 1 - 1.33

Serialization - 11


- Like so:

```
[Serializable()]
class BasicAccount : Account, ISerializable
{
    private int overdraft;

    public BasicAccount (SerializationInfo info, StreamingContext context) : base (info, context) {
        overdraft = (int)info.GetValue ("Overdraft", typeof(int));
    }

    public override void GetObjectData (SerializationInfo info, StreamingContext context)
    {
        base.GetObjectData (info, context);
        info.AddValue ("Overdraft", overdraft);
    }
}

```



33

Title of Topic: Topic 1 - 1.34


Serialization - 12

- And:

```
[Serializable()]
class ExtendedAccount : Account, ISerializable
{
    private double interestRate;

    public ExtendedAccount (SerializationInfo info, StreamingContext context) : base
    (info, context) {
        interestRate = (double)info.GetValue ("Interest Rate", typeof(double));
    }

    public override void GetObjectData (SerializationInfo info, StreamingContext context)
    {
        base.GetObjectData (info, context);
        info.AddValue ("Interest Rate", interestRate);
    }
}
```




34

Title of Topic: Topic 1 - 1.35

Serialization - 13

- With this, we're ready to save our objects, which we'll do whenever any state changes in the main program.



35

Title of Topic: Topic 1 - 1.36

Using Serialization - 1


- Serialized data is **binary**, and it uses a different kind of system.
- Usually we do this with a base stream and a binary formatter object, like this:

```
private void saveData() {
    Stream output;
    BinaryFormatter bf = new BinaryFormatter();

    output = File.Open ("atmdata.atm", FileMode.OpenOrCreate);

    bf.Serialize (output, myAccounts);

    output.Close();
}
```



36

Title of Topic: Topic 1 - 1.37

Using Serialization - 2


- And when we load it in, we deserialize via a similar formatter like so:

```
private void loadData() {
    Stream output;
    BinaryFormatter bf = new BinaryFormatter();

    output = File.Open ("atmdata.atm", FileMode.Open);

    if (output.Length != 0)
    {
        myAccounts = (List<Account>)bf.Deserialize (output);
    }

    output.Close();
}
```




37

Title of Topic: Topic 1 - 1.38

Using Serialization - 3

- Now, all we need to do is call these functions at the appropriate time.
- We'll call loadData at the load event, and if there is nothing that comes out of it we'll populate our usual test data:



38

Title of Topic: Topic 1 - 1.39

Using Serialization - 4


```
private void frmCashMachine_Load(object sender, EventArgs e)
{
    enteredText = "";
    enteredPin = "";
    enteredNumber = "";

    loadData();

    if (myAccounts == null) {
        myAccounts = new List<Account>();

        addBasicAccount ("20", "1234", 10000);
        addBasicAccount ("30", "4321", 55000);
        addExtendedAccount ("40", "5555", 1000, 1.5);
        addExtendedAccount ("50", "9966", 150000, 2.0);
    }

    inputPermitted = true;
}
```




39

Title of Topic Topic 1 - 1.40

Using Serialization - 5

- And finally we add the saveData call when we manage a successful deposit:

```
if (ret == true)
{
    lblOutput.Text = "Transaction successful. You have a " + current.queryType()
+ " account";
    saveData();
    ...
}
```




40


Title of Topic Topic 1 - 1.41

Using Serialization - 6

- Run your program, try some transactions, and then shut it down.
- Start it up again, and you should find the state of your data is the same as how you left it.
- It needs a little bit of extra class setup to make this work, but the good news is that you only need to do it for classes that you want to save.
- Nothing else needs the extra work.



41



Awarding
Great British
Qualifications

Topic 11 – File IO

Any Questions?

42
