Inft2012 Application Programming

Lecture 8

Lists, file input and output

List collections

- A List is something like an array – its elements can be accessed by its name and an appropriate index – but
  - its size isn't fixed and isn't declared; you don't need to know in advance what size to make it
  - elements can be dynamically added and removed
  - there are a number of very helpful methods
- Last week we looked at the ListBox and ComboBox controls; the Items collections of these controls are a form of List
- When we declare a List, we have to give its type as an argument; we use angle brackets for this purpose

Some List methods

- Once a List is declared, eg
```
List<string> lisName = new List<string>();
```
- You can add items to the end:
```
lisName.Add("Tan Han Kee");
```
- You can insert an item at a specified index:
```
lisName.Insert(3, "Simon");
```
- You can remove specified items:
```
lisName.Remove("Rumpelstiltskin");
```
- You can remove the item at a specified index:
```
lisName.RemoveAt(2);
```
- You can find its size:
```
iSize = lisName.Count;
```


More List methods

Some higher-level List methods . . .

- Clear() – remove all elements
- Clone() – return a copy of the List
- Contains(item) – whether item is in the List
- IndexOf(item) – index of first occurrence of item
  – also LastIndexOf(item)
- Reverse() – reverse the order of the elements
- Sort() – sort the elements – you might explore how

Cycling through a List

- If lisName is a List, lisName.Count tells us how many elements are in the List
- We could use this in a for loop to access each element of the List in turn by its index
- However, there's an easier way:

```
foreach (string sName in lisName)
{
    // process sName
}
```


One more example

- One more example to show how a list can sometimes be easier to work with than an array
- Imagine that for some reason we want to move the first element of a list to the last place
- With an array, this would mean explicitly moving each element to the next position down, except for the first element, which is moved to the last position
- With a list . . .

```
lisName.Add(lisName[0]);
lisName.RemoveAt(0);
```

- See the List of string form in Lec8Demo

ArrayLists

- Arrays and lists are restricted to items of the same type (eg an array of int, a list of string, etc)
- The ArrayList overcomes this limitation
- Like a list, an ArrayList is a collection, but it is a collection of *objects*
- As a collection, it has many of the nice features of lists that we have just been exploring

Using ArrayLists

- Before you can declare an ArrayList, you need to add `using System.Collections;` at the start of your code
- There is a disadvantage to ArrayLists: although we can add ints, doubles, bools, strings, and more to the same list, every element is stored as an object
- If we actually need to use different types of elements in different ways, we need to know what type of element each one was before it was added, so that we can convert it back to that type
- Of course, if we use an ArrayList to hold only doubles, for example, we always know that to use an element numerically we must convert it to a double

File storage

- Persistent data is data that remains in storage when the program isn't running
- In all the programs we've written so far, all the data ceases to exist when the programs aren't running
- Files are the most common form of persistent data
- Programs can read files as their input
- Programs can write files as their output
- What software do you use that does this?

Files and streams

- Computer files were originally stored on punched paper tape or magnetic tape
- An unavoidable feature of a file was that the program had to access each element in turn; even if it only wanted the ones near the end, it first had to go through all the ones at the start and in the middle
- Also, you couldn't write to a file while partway through reading it. The best you could do was read the whole file then write the whole file – though you could read from one and write to another at the same time
- This leads to the concept of a file as a 'stream' of data, a steady flow past the program

Stream-related classes

- File-handling in C# is typically done with various stream-related classes, such as StreamReader and StreamWriter, which read and write text files
- There are other associated classes, such as File and FileStream
- Using them is a little tricky until you get used to them
- To read from a file, you associate the file name with a StreamReader, then Open the StreamReader, read the contents of the file from it, and Close the StreamReader
- The process is similar for writing, except that there are more steps involved to specify whether the file is being opened for writing or appending

File I/O often throws exceptions

- File I/O (input and output) can generate many errors
  - File not found
  - Input not of right type
  - File already exists
  - etc
- Therefore file I/O will always be found in the context of try and catch
- The simplest action in a catch is to echo the message that would have been written if the program had crashed
- More helpful messages make for more robust applications

File locations

- The default file location for a Visual Studio application is the Debug/bin folder in the application generated by the folder
- For program development and debugging, put the relevant files there
- Other locations can be specified by relative paths . . .

  ..\..\filename (two folders up)
- or absolute paths . . .

  c:\Uni\Inft2012\CsharpProjects\Lec8Demo\filename

  – but how often will this fail?

Literal file paths in the program

- The backslash character is part of a path name in Windows systems . . .
- but it's also the symbol in a string that means the following character is something special . . .
- "Your name is . . .\n\tRumpelstiltskin!"
- so literal paths will pose some difficulty
- One approach is to precede every backslash in the path with another backslash:

  sFilename = "c:\\Uni\\Inft2012\\CsharpPrograms\\file";
- Another is to precede the whole string with @, which means treat every character literally:

  sFilename = @"c:\Uni\Inft2012\CsharpPrograms\file";

File Dialog controls

- C# provides standard file location dialog boxes, which return a DialogResult (typically OK or Cancel), but also have the path and name of the selected file in a FileName property
- Unlike message boxes, these boxes need to be added to a form from the toolbox before they can be used
- Like timers, these boxes go in the component tray beneath the form
- They don't actually do the opening or saving: they locate the files so that the program can then open them or save them

Simple file I/O demo

- The CaesarShift form demonstrates text file input and output at about its simplest – use the file Berrybrew.txt
- The way it reads and writes is a real cop-out – reading a whole file into a string at one hit, and writing a whole file from a string at one hit
- Real text files are typically too big to handle in this way
- A more common approach would be to read and write a character at a time or a line at a time
- As always, examine the program very carefully, making sure you understand every single aspect of it –
- Learning to write code is a lot easier once you've learnt to read it!

Rainfall with file handling

- Last week we mentioned that the rainfall data would normally be loaded from a file
- Now we can do that – and we get to choose any year from 2003 to 2017
- We can also save any changes we make to the data, by writing a new copy of the file
- Note that the rainfall is in comma-separated value files; these are created from an Excel spreadsheet, and care has to be taken to preserve their format when writing
- (One way to see what they look like is to open them with Notepad or some other text editor)

Points to note

- The OpenFileDialog is used to locate the file, but the file is then opened by subsequent statements
- There are often many ways to open files
- This demo opens a StreamReader using File.OpenText; File is a superclass of StreamReader and StreamWriter
- It opens a StreamWriter using a FileStream rather than a file name; FileStream is a superclass of StreamReader and StreamWriter
- When you need to open files, read a book or online help carefully so that you know what's going on, then follow a suitable example

Another point to note

- This demo shows how easy it can be to keep a check on whether a file needs saving
- There's a simple boolean value that's set to false when a file is loaded, true when a file is changed, and false when a file is saved
- When the user tries to load a file, the program issues a warning if the current file has been changed and not saved
- The system could deal with more cases, but this gives a good idea of how this checking works

Binary files

- The files we've seen so far are all text files – even though the rainfall ones have a csv extension
- Like most programming languages, C# can also load and save pure binary files, files whose contents are stored in their internal representation rather than as text equivalents
- A sequence of numbers can be stored as a sequence of numbers – which means there's no need to bother with commas, spaces, etc
- See how the rainfall demo uses binary files – and try looking at those files with Notepad or another plain text editor

Binary files and FileStream

- Binary files aren't as simple to open as text files
- StreamReader and StreamWriter are a very nice simplification to make it easy to use text files
- While StreamReader and StreamWriter can be instantiated and initialised on the basis just of a file name . . .
- BinaryReader and BinaryWriter both have to be based on FileStream objects . . .
- And it's the FileStream that's based on the file name

Binary files and data types

- When you write a value to a file using BinaryWriter, it knows what type the value is, so it knows how to write it
- (Look at the many overloaded versions of BinaryWriter's Write method)
- When you read a value from a file using BinaryReader, you (the programmer) have to know what sort of value to expect . . .
- And you have to use the appropriate Read method
- (Look at BinaryReader's many different Read methods)

Mixing the data types

- If you have an object whose members include strings, integers, doubles, booleans, and more . . .
- You can write them to a binary file and read them from a binary file
- But you have to be absolutely consistent in the types you use and the order in which you read and write them
- Effectively, you're back to parsing the input, working out which bits serve what purposes

Recognising end of text file

- The file-reading examples we've seen here don't need to look for the end of the file (Why not?)
- In most circumstances it's very important to check for the end of file when doing any reading
- With a text file, when the file has been completely read the next line (or string) will be null (a reserved word, not the same as "")
- What sort of loop should you use when reading from a file whose size you don't know in advance?
- The 'Text file line by line' form shows an example of reading lines until reaching end of file; use it with PigChapter1.txt

Recognising end of binary file

- This doesn't work with a binary file
- However, a stream has a couple of useful properties:
- Length, the length of the stream
- Position, the position that reading has reached in the stream
- These can be used as follows:

```
iLgth = inFile.BaseStream.Length;
while (inFile.BaseStream.Position != iLgth)
{
    // Read and process the next item
}
```