Lecture 33 — Files & Streams

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

July 28, 2016

Acknowledgments: W.D. Bishop

Files and File Systems

Files permanently store large amounts of data.

Operating systems implement a filesystem to manage files stored on a hard disk (or other types of storage media).

Filesystems typically support the following:

- Creating new files
- Reading from a file
- Writing to a file
- Appending to existing files
- Renaming existing files
- Deleting existing files
- Checking for the existence of a file

Streams

File I/O and console I/O are very similar.

It is possible to define I/O methods such as `ReadLine( )` and `WriteLine( )` that work with both file I/O and console I/O.

Streams are an abstraction for data flow that can be applied to both file I/O and console I/O.

A stream denotes the flow of data from a source to a destination.

In this course, we have already used streams:
- `Console.Out` is the stream for console output.
- `Console.In` is the stream for console input.

File I/O Fundamentals

The file I/O classes are defined in the `System.IO` namespace.

Output streams are created using the `StreamWriter` class.

Input streams are created using the `StreamReader` class.

The `StreamWriter` and `StreamReader` classes provide sequential access to text files: each line must be written or read in sequence.

Files are assumed to store text data.

File Output Example

This code outputs the text “Hello World” to a file named “output.txt”.

```
StreamWriter outStream = null;
string filename = "output.txt";

try
{
    outStream = new StreamWriter( filename );
    outStream.WriteLine( "Hello World" );
    outStream.Close( );
    Console.WriteLine( "File successfully created" );
}
catch( Exception e )
{
    Console.Write( "Unable to create file! Reason: " );
    Console.WriteLine( e );
    return;
}
```

If output.txt didn’t exist before, it creates that file.

File Input Example

```
StreamReader inStream = null;
string filename = "input.txt";
string line;

try
{
    inStream = new StreamReader( filename );
    do
    {
        line = inStream.ReadLine( );
        Console.WriteLine( line );
    }
    while( line != null );
    inStream.Close( );
    Console.WriteLine( "File successfully read" );
}
catch( Exception e )
{
    Console.Write( "Unable to read file! Reason: " );
    Console.WriteLine( e );
    return;
}
```

File Input Example

The file input example code reads a specified file (input.txt) into memory (variable `line`), one line at a time.

That line of text is then printed out to the console.

Note in both input and output examples the use of `try-catch` blocks. File-related operations may frequently result in Exceptions.

Inputting Formatted Data

The `ReadLine( )` method reads complete lines of data.

However, some applications use numeric input data or other formatted data fields stored as text.

If the input data consists of non-string data types represented as text, the lines of input must be split into individual strings and parsed.

There are a couple of things to consider:

- How will the data be used, stored, manipulated, etc.?
- What errors should be detected, corrected, ignored, etc.?

Reading Data Example

Given the following class declaration:

```
public class Employee
{
    ulong identifier;
    string surname;
    string givenName;
    double salary;
}
```

How would you build a program to read this data from a file and display the data on the console?

Step 1: File Format

Before coding anything, consider the format of the input file.

Let's assume a valid input file resembles the following:

```
3 Phaneuf Dion 6500000
81 Kessel Phil 6000000
12 Connolly Tim 5500000
8 Komisarek Mike 5500000
24 Liles John-Michael 4550000
```

This is an easy file to parse since we can use the `Split( )` method of the `string` class to split lines into arrays of strings.

Step 2A: Updating `Employee`

Next step: to make importing of data efficient, we should add a constructor to `Employee`.

```
public Employee( ulong id, string sname,
                 string gname, double s )
{
    identifier = id;
    surname = sname;
    givenName = gname;
    salary = s;
}
```

Step 2B: Updating `Employee`

Next step: to make output human readable, implement `toString`.

```
public override ToString( )
{
    string name = surname + ", " + givenName;
    return( string.Format( "{0,-4}: {1,-30} {2,12:C}",
            identifier, name, salary ) );
}
```

Step 3: Write the Main Method

```
static void Main( )
{
    StreamReader inStream = null;    // This is the filehandle for the input file
    string filename = "input.dat";   // This is the filename for the input file
    string line = null;              // This is a variable to refer to a single line of input
    string[] fields = null;          // This is an array of strings to store the individual fields
    int counter = 0;                 // This is a counter to keep track of the number of records read
    Employee emp = null;             // This is a variable to refer to an employee record

    Console.WriteLine( "\nDatabase File Reader\n" );
    inStream = new StreamReader( filename );
    do                    // Reads lines of input
    {
        line = inStream.ReadLine( );
        if( line == null )
        {
            break;
        }
        counter++;
        Console.WriteLine( "Record [{0}]:", counter );
    }
    while( true );
    inStream.Close( );
    Console.WriteLine( "\nSuccessfully read {0} records", counter );
}
```

Step 4: Modify the Program to Parse Data

Replace the content of the `do-while` loop with:

```
line = inStream.ReadLine( );
if( line == null )
{
    break;
}
counter++;
fields = line.Split( );
emp = new Employee(
                   UInt64.Parse( fields[0] ),
                   fields[1],
                   fields[2],
                   double.Parse( fields[3] )
                   );
Console.WriteLine( "Record [{0}]:", counter );
Console.WriteLine( "\t{0}", emp );
```

Step 5: Add Error Checking

To add error detection, simply enclose the file I/O in a `try` statement and add a corresponding `catch` statement.

Adding the error checking code is left as an exercise.

Questions to Ponder

How would you deal with input data that includes spaces?

You could use commas or another special character to delimit fields.
...but how would this change your parsing...?

What would you do if you needed to read a file containing a fixed number of records into an array of records?

- Read the file twice: once to determine the number of records and once to store them?
- Store the number of records at the top of the input file?
- Set a maximum number of entries?
- Re-size the array when necessary?
- Use a collection?

Field Delimiters

The `Split( )` method provides the ability to specify one or more delimiters.

When the `Split( )` method is called, a character array containing a set of delimiters can be passed to the method.

The easiest way to create an array of delimiters is to create a string of the delimiters and then convert the string to a character array.

A string s can be converted to a character array using the ToCharArray( ) method:

```
  ",.&".ToCharArray( )
```

Field Delimiter Example

What happens when we try to split `line` below?

```
string line = "A,B;C D;E,F";
char[] delimiters = new char[] { ' ' };
string[] fields;

Console.WriteLine("\nSplitTest1\n");

fields = line.Split( delimiters );
```

We get two strings as output:

"A,B;C"
"D;E,F"

Field Delimiter Example 2

What happens when we try to split `line` below?

```
string line = "A,B;C D;E,F";
char[] delimiters = new char[] { ',' };
string[] fields;

Console.WriteLine("\nSplitTest1\n");

fields = line.Split( delimiters );
```

We get three strings as output:

"A"
"B;C D;E"
"F"

Field Delimiter Example 3

What happens when we try to split `line` below?

```
string line = "A,B;C D;E,F";
char[] delimiters = new char[] { ';' };
string[] fields;

Console.WriteLine("\nSplitTest1\n");

fields = line.Split( delimiters );
```

We get three strings as output:

"A,B"
"C D"
"E,F"

Field Delimiter Example 4

What happens when we try to split `line` below?

```
string line = "A,B;C D;E,F";
char[] delimiters = new char[] { ' ', ',', ';'  };
string[] fields;

Console.WriteLine("\nSplitTest1\n");

fields = line.Split( delimiters );
```

We get six strings as output:

"A"
"B"
"C"
"D"
"E"
"F"

Terminology Review

In our discussion of file I/O, we have introduced a number of important database terms:

1. A file is form of long-term, persistent data storage
2. A database file is a collection of records
3. A record is a collection of fields separated by delimiters
4. A field is a group of bits (or characters)
5. A delimiter is a special character used to separate fields

A detailed discussion of databases is beyond the scope of this course.

Comments on C# File Security

Some practical information if you are trying to work with files in C#.

By default, a C# program does not have permission to create or modify a file if the program resides on a network drive.

For example, all of the file I/O example programs provided in the lecture notes will NOT run on a remote (network) drive.

There are two ways to compile and test programs that use file I/O:

1. Copy the executable program to a local directory (i.e., C:\TEMP) and execute it there.
2. Include attributes / code fragments to modify the default file I/O permissions or to modify the security policy

Files and Directories

C# also provides a couple of classes with static methods to let you work with files and directories.

They are called `File` and `Directory`, respectively.

For those who like a challenge, write a program to display the entire directory structure of a filesystem.