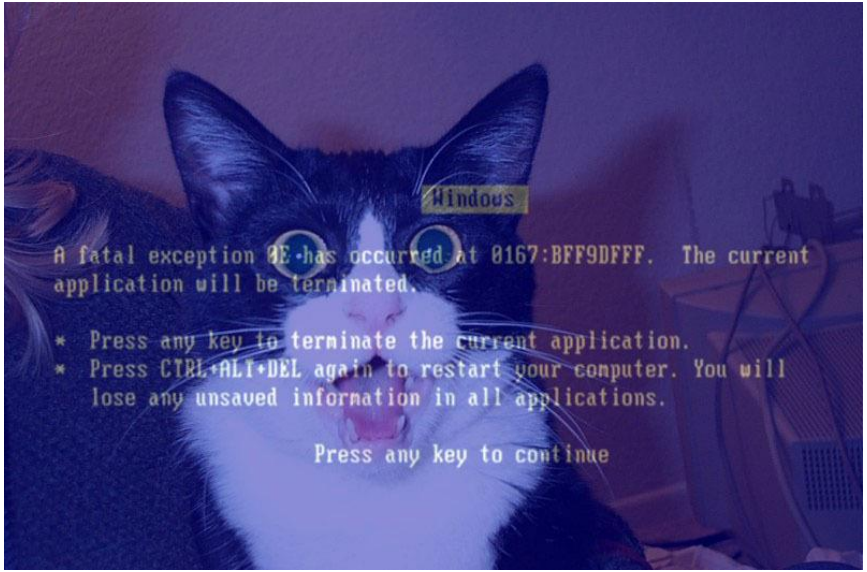
EXCEPTIONS, TRY CATCH FINALLY, ASSERT, TESTING, AND FILE I/O



BME 121 2016

Jeff Luo

Topics

- Exceptions
- Try Catch
- Assert
- Basic Testing Framework
- File I/O
- Try Catch Finally
- Flush

Exceptions

- A mechanism for crashing a program if it is doing something invalid.
- Eg:
  - System.DivideByZeroException
    - If we are dividing by 0 somewhere in the program.
  - System.IndexOutOfRangeException
    - If we are accessing an array with an index that’s outside its bounds
- Programs we write automatically issue some Exceptions, but we can also create and issue our own.
- All Exceptions inherit code from class System.Exception

Custom Exceptions

- To create a custom exception, define an Exception class:

```
public class MyCustomException : Exception
{
    // all three versions of the constructor must be defined exactly as follows:
    public MyCustomException() {}
    public MyCustomException(string message) : base(message) {}
    public MyCustomException(string message, Exception inner) : base(message, inner) {}
}
```

Exceptions

- Issuing an exception:

```
throw new MyCustomException();
// Unhandled Exception: ConsoleApplication.MyCustomException: Exception of type
'ConsoleApplication.MyCustomException' was thrown.
```

- Issuing an exception with a custom message:

```
throw new MyCustomException(“your program is crashing!”);
// Unhandled Exception: ConsoleApplication.MyCustomException: your program is crashing!
```

- Issuing an exception with a message and an additional exception:

```
throw new MyCustomException("woops", new InvalidOperationException() );
// Unhandled Exception: ConsoleApplication.MyCustomException: woops --->
System.InvalidOperationException: Operation is not valid due to the current state of the object.
```

  - Used if one exception is caused by another exception.

Using Exceptions

- Don’t use exceptions if it is something your code anticipates:
  - Bad user inputs of any kind → offer the user some way of re-entering the values instead of crashing the program.

- Issue exceptions when a fundamental assumption made by the code is found to be false.
- Example:
  - Method `Fib(int x)` is mathematically undefined for negative x values, throw an exception if x is negative.
  - Method `double[,] MatrixProduct(double[,] x, double[,] y)` calculates the product of the matrices x and y. By math definition, the number of columns of x must be the same as the number of rows of y. If this isn’t the case, throw an exception.

Practice 1

- Using MatrixProduct.cs as a starting point:
  - Create a custom exception class called MatrixSizeIncompatibleException, ensure all 3 versions of the constructor are defined.
  - In method MultiplyMatrix, throw a custom exception with a message indicating the dimensions of each matrix and why the two matrices are incompatible.
  - In Main, create an additional matrix that is incompatible with Matrix x or y, and call your method to make sure it throws the exception.

Stack Trace

- When an Exception occurs, we get some information as to where it happens in the code.
- In this example, Main() calls A(), which calls B(), which calls C(). C throws an exception.

Run StackTrace.cs. Output is:

```
Unhandled Exception: System.Exception: Custom crash!
   at ConsoleApplication.Program.C()
   at ConsoleApplication.Program.B()
   at ConsoleApplication.Program.A()
   at ConsoleApplication.Program.Main(String[] args)
```

Try / Catch

- A lot of code can throw exceptions, including custom exceptions we create. By default, exceptions crash the program.
- Instead of crashing the program, we can write some code which will gracefully return the program to normal operation using try / catch:

```
try
{
    // run any code that might throw an exception
}
catch (Exception e)
{
    // if an exception occurs, code within catch will run
}
```

Try / Catch

- Code within a `catch` statement will execute for any matching exceptions thrown within `try`.

```
string x = "word";  int count = 0;
try
{
    while(true)
    {
        Console.Write("\rCurrently at loop {0}", ++count);
        // take two copies of the string x, put them together and store it in x
        x = x + x;
    }
}
catch(OutOfMemoryException e) {
    Console.WriteLine("YO something bad happened.");
}
```

Try / Catch

- Multiple Catch is also allowed:

```
try
{
     // some code
}
catch (CatException ce)
{

}
catch (DogException de)
{

}
// …
```

- The most general exception to catch is always

catch (Exception e)

- Generally, start any try catch code block with Exception e and then customize it into multiple catch if you need it.

Practice 2

- Using MultiCatch.cs as a starting point, investigate the possible exceptions that can be thrown by the code in the try block, and add a `catch` for each of these exceptions. Within the `catch`, display an appropriate message.

Assert

- Sometimes during the development of a program, we want to make sure some property is true in our code.
- We use assert command to perform these tests:

```
Debug.Assert( test );          // test evaluates to a bool
```

- Note: put `using System.Diagnostics;` up at the top of the program.
- For example, in Max(int[] x) we might want to assert that x has at least 1 element:

```
int Max(int[] x) {
        Debug.Assert( x.Length > 1 );
        // code for finding the max
}
```

Assert

- If the test passes, program continues executing all following lines of code.
- If the test fails, a System.Diagnostics.Debug+DebugAssertException is thrown by the Assert method:

```
Debug.Assert( 5 > 20 );
```

```
Unhandled Exception: System.Diagnostics.Debug+DebugAssertException:
   at System.Environment.GetStackTrace(Exception e, Boolean needFileInfo)
   at System.Environment.get_StackTrace()
   …
   at ConsoleApplication.Program.Main(String[] args)
```

Assert

- Two main versions of the Assert method:
- Simply ensure the test passes:

```
Debug.Assert( test );
```

- Ensure the test passes, but provide a message if it fails:

```
Debug.Assert( test, “The array must have at least 1 element.” );
```

```
// Console output:
Unhandled Exception: System.Diagnostics.Debug+DebugAssertException:
The array must have at least 1 element.
```

Note: you can’t catch `System.Diagnostics.Debug+DebugAssertException`

Using Assert to Test Software

- We can use Assert to ensure that a method returns an output which matches our expected output, and if it doesn't, it immediately crashes the program with a message that helps us identify which test failed:

```
// in Main(), testing a method called M():
Debug.Assert(  M(input1) == expectedOutput1  , “Test 1 failed”);
Debug.Assert(  M(input2) == expectedOutput2  , “Test 2 failed”);
// …
```

- Each assertion ensures 1 single test case passes

Basic Testing Framework

- Combining Try / Catch and Assert in a loop, we can create a basic testing framework:

```
// Define a test set using two parallel arrays
int[] input = { 1, 2, 3, 4 };
int[] expectedOutput = { 7, 8, 9, 10 };

// Use a loop to execute all of the test cases
for(int i = 0; i < input.Length; i++) {
     try  {
          Debug.Assert( PlusSix(input[i]) == expectedOutput[i] );
     }
     catch (Exception e) {
          Console.WriteLine("Test {0} failed", i + 1);
     }
}
```

Practice 3

- Design a set of tests with at least 6 test cases for method `int Max(int[] x)`
  - Remember: test cases should examine both the expected behavior of the method (finding the max value of an array), and edge cases (length 0 array, and no array)

Test Case	Input	Expected Output
1	{0, 1}	1
2	{3, 3, 3}	3
3		
4		
5		
6		

- Using TestMax.cs as a starting point, add some code in Main which tests the Max method using your test cases.

Practice 4

- Given the test set on the right, implement a method which solves this test set using TestDrivenDevelopment.cs as the starting point.

- As you write the code for the method, run the program to see how many tests still needs to pass.

Test Case	Input	Expected Output
1	0	{}
2	1	{0}
3	3	{0,3,6}
4	2	{0,2}
5	4	{0,4,8,12}
6	-1	ArgumentOutOfRangeException
7	-30	ArgumentOutOfRangeException

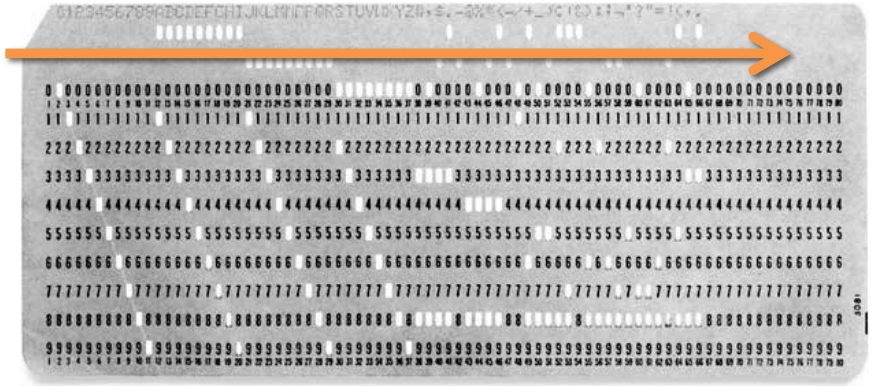
Assert and Compilation

- Note: We have been compiling our programs using a default compilation mode where Asserts are compiled, this is okay for development purposes.

- C# by standard will ignore Debug.Assert() calls when compiling in Release mode.
- Assert helps us discover bugs in our programs. Failures in release code (programs given to customers) indicate that a bug is missed. We don't really want our users to be debugging our programs, thus C# ignores Assert calls in Release mode, allowing us to mix code used for testing and actual code for the user.

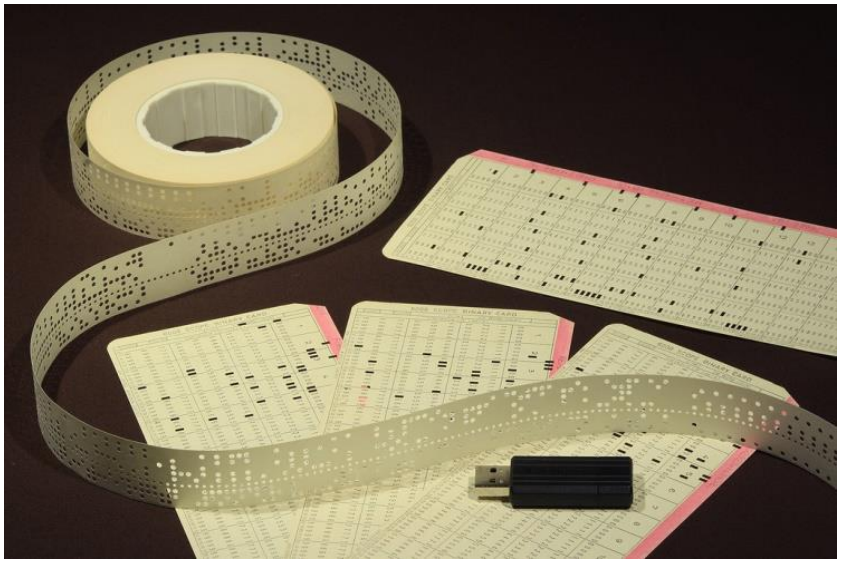
Bytes and Streams

- For historical reasons, computer systems treat practically all input and output of continuous data sources as streams
- A stream contains a sequence of data divided up into unit pieces
  - IBM Punch Card: each unit can hold 1 numerical digit
  - In modern day: 1 byte per unit of data
- A stream maintains a position "cursor", and this moves in a forward direction whenever we read or write to the stream



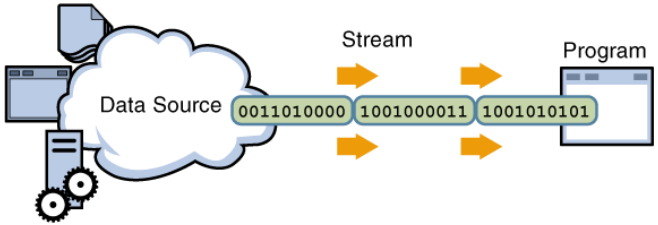
Streams

- **Punch Tape** was invented in part by inspiration from river streams. Instead of fixed amounts of data stored on each Card, a Tape provided a continuous stream of data. Practically it was easier to deal with than tracking and sorting thousands of cards in a Punch Card deck.
- At the lowest level, Streams allow the following operations:
  - **Read Byte** (returns 1 Byte)
  - **Read Bytes** (returns array of Bytes)
  - **Seek** (move cursor to a desired point)
  - **Push Back Bytes** (“undo” of Read Bytes)
  - **Peek** (inspect the next few Bytes without reading it)
    Write Byte
  - **Write Bytes**
  - **Skip Byte(s)** (fast forwards the write position)
  - Very similar to music players!
- In C#, we deal with streams at a much higher level!

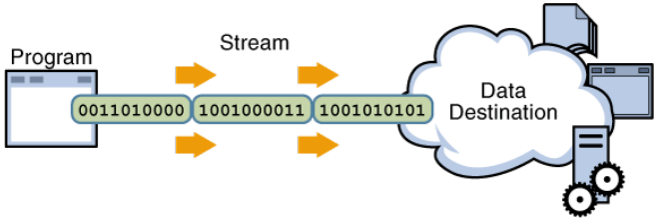


Types of Streams

- Input Only Streams



- Output Only Streams

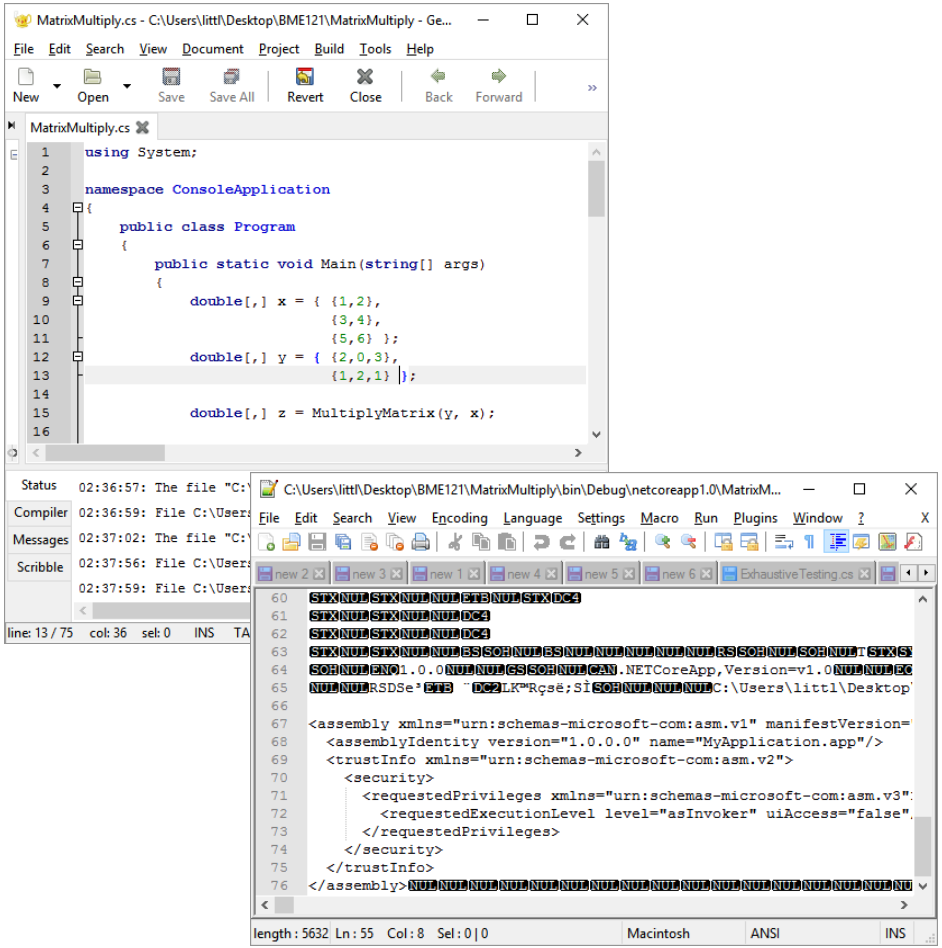


- Input and Output Streams
  - Eg Console

Computer Files

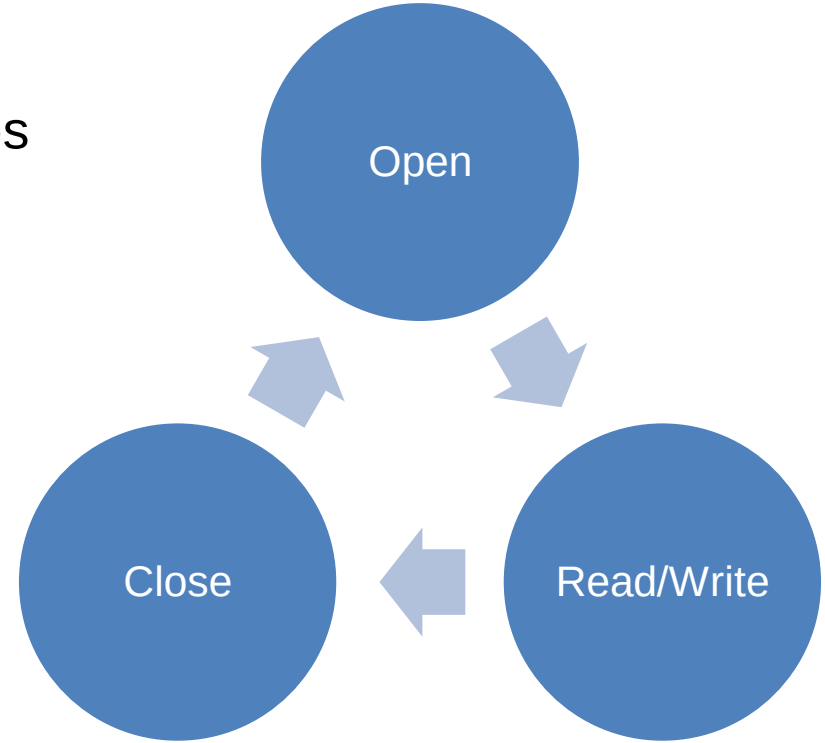
- A File is technically a logical unit of a sequence of data
- Consists of either:
  - **ASCII/Unicode Text data**
    - Which notepad can open and interpret as human readable text
  - **Binary data**
    - Which only the computer can understand, ie they are programs
  - **A mix of both**
    - Open up a C# exe file in notepad!

- A Computer Folder is actually a special file which contains the information of where it's subfiles are stored in a storage medium



C# File Operations

- Files are both a source of data, and a destination of data
- Must be **Opened** before it is used
- Must be **Closed** before program terminates



C# File Operations

- The System.IO namespace contains
  - Types that allow reading and writing to files and data streams
  - Types that provide basic file and directory support

- The FileStream class orders a sequence of bytes which can be arbitrarily accessed, and read or written to
  - Used by itself, it reads and writes bytes to a file

- The StreamReader and StreamWriter classes provide sequential access to **text** files
  - They can be wrapped around the FileStream to create a file, and read/write from/to it
  - Each line of the file must be written or read in sequence

- Input streams are created using the StreamReader class
- Output streams are created using the StreamWriter class

Reading Files

1. Create an object of `FileStream` and bind it to a specific file, using Open mode and Read access, this will open the file for reading:

```
FileStream inFile = new FileStream(@"myfile.txt", FileMode.Open,
FileAccess.Read);
```

2. Create an object of `StreamReader` to interpret the file as a text file:

```
StreamReader inStream = new StreamReader(inFile);
```

3. Use the `StreamReader` object to read the file one line at a time:

```
string line = inStream.ReadLine();
```

4. Remember to close the stream and the file:

```
inStream.Dispose();
inFile.Dispose();
```

Note: "myFile.txt" assumes myFiles.txt is in the same directory as yourProgram.cs file

Reading Files

- Read a File, display it's contents, and count and display the number of lines in the file:

```
using System;
using System.IO; // Don't forget to import System.IO

// in Main():
// Create an object of FileStream and bind it to a specific file
FileStream inFile = new FileStream(@"myfile.txt", FileMode.Open, FileAccess.Read);
StreamReader inStream = new StreamReader(inFile);

Console.WriteLine("Content of the file:");
int counter = 0;
while(!inStream.EndOfStream)
{
   // read a line of text from the stream, then display that line of text
   Console.WriteLine( inStream.ReadLine() );
   counter++;
}
Console.WriteLine("Number of lines = {0}", counter);

inStream.Dispose(); // close the stream
inFile.Dispose(); // then close the file
```

Files

```
FileStream inFile = new FileStream(@"myfile.txt", FileMode.Open,
FileAccess.Read);
```

@ marks the string as a verbatim string literal - anything in the string that would normally be interpreted as an escaped character sequence is ignored.

So `"C:\\Users\\Rich"` is the same as `@"C:\Users\Rich"`

For windows:

```
@"c:\users\USERID\desktop\bme121\folder\myfile.txt"
```

For mac:

```
@"\users\USERID\desktop\bme121\folder\myfile.txt"
```

Writing Files

1. Create an object of `FileStream` and bind it to a specific file, using Create mode and Write access, this will create (or replace) the file for writing:

```
FileStream outFile = new FileStream(@"myfile.txt", FileMode.Create,
FileAccess.Write);
```

2. Create an object of `StreamWriter` to write text to the file:

```
StreamWriter outStream = new StreamWriter(outFile);
```

3. Use the `StreamWriter` object to write to the file:

```
outStream.Write(“Some text”);
outStream.WriteLine(“Some text”);
```

4. Remember to close the stream and the file:

```
outStream.Dispose();
outFile.Dispose();
```

Microsoft Docs on
FileMode: link
FilleAccess: link

Writing Files

- Create/replace a file, where the contents is 20 random numbers:

```
using System;
using System.IO; // Don't forget to import System.IO

// in Main():
// Create an object of FileStream and bind it to a specific file
FileStream outFile = new FileStream(@"myfile.txt", FileMode.Create, FileAccess.Write);
StreamWriter outStream = new StreamWriter(outFile);

Random rng = new Random();

// Write 20 random numbers to a file
for(int i = 0; i < 20; i++)
{
    outStream.WriteLine(rng.Next(0, 30));
}

outStream.Dispose(); // close the stream
outFile.Dispose(); // then close the file
```

Try / Catch / Finally

```
try {
    // code that might throw exceptions
}
catch (Exception e)
{
    // code that runs if an exception occurs
}
finally
{
    // code that will always run after the try block,
    // even if exceptions are thrown
}
```

Optionally used to clean up variables or objects created in try

File I/O Recommended Style

```
FileStream inFile = null;
StreamReader inStream = null;
string fileName = @"myfile2.txt";
try
{
   // Note: this can throw FileNotFoundException
   inFile = new FileStream(fileName, FileMode.Open, FileAccess.Read);
   inStream = new StreamReader(inFile);
   // do stuff with file
}
catch (FileNotFoundException e)
{
   // code that should run if “myfile.txt” doesn’t exist
}
finally
{
   if(inStream != null) inStream.Dispose(); // close the stream
   if(inFile != null) inFile.Dispose(); // then close the file
}
```

Open and use a file in `try`, handle exceptions in `catch`, and close the file in `finally`.

Writing Files – Flushing

- In standard use, **StreamWriter** doesn't actually write a file in the computer until outStream.**Dispose()** is called and executed.
- All the data is queued up in the Stream and then "Flushed" into a file:
  - outStream.**Flush()** manually asks the StreamWriter to write out whatever is already in the queue to the file. Any new data queued in the stream afterwards will append the file when Flush() is called again
  - outStream.**Dispose()** calls outStream.**Flush()** behind the scenes

- If you are writing out a very large file with many lines of data then it becomes important to manage the active memory used by your program. So, flush things periodically using **Flush()** to move all the queued writing data from memory to disk

- Also could set **AutoFlush** property to true, then the stream will flush after every call to Write() or WriteLine()
  - outStream.AutoFlush = true;