Exceptions and File Processing

Exceptions

On occasion a segment of code in a program may result in an error condition, such as an attempt to divide by zero, an array index out of bounds, failure to open a file when requested, etc.

Careful program design can help minimize the number of exceptions that occur; however, occasionally they do happen.

In this material we will show how you can handle exceptions when they occur, rather than letting the program just terminate. This is done with try and catch.

try

Suppose a program segment has a chance of causing an exception. That code can be placed in a try segment.

```
try
{
   //code that might cause the error.
}
```

catch

If the code in the try does indeed result in an error, an exception is thrown, which can then be handled in a catch.

```
catch
{
   //process the error.
}
```

```
while (M!=0 || N!=0)
{
        Console.WriteLine("Enter two integers.");
        try
        {   Console.Write("First Integer: ");
            M = Int32.Parse(Console.ReadLine());
            Console.Write("Second Integer: ");
            N = Int32.Parse(Console.ReadLine());
            Console.WriteLine("The quotient is {0}.", M / N);
        }
        catch (DivideByZeroException)
        {  Console.WriteLine("Attempt to divide by zero");
            continue;
        }
        catch (FormatException)
        {
            Console.WriteLine("Both must be integers:");
            continue;
        }
 }
```

Run the Program

Let’s take a look at this program segment, first running it with the try, and then running it with the try and catch removed.

throw an exception

```
int N = 2;
Exception NegativeValueException=new Exception("No Negatives");
int M=0;
do
{
    try
    {
        Console.Write("Enter a positive number: ");
        M = Int32.Parse(Console.ReadLine());
        if (M < 0)
            throw (NegativeValueException);
        Console.WriteLine("You entered " + M + " Hit Enter to continue.");
        Console.ReadLine();
    }
```

throw example (cont.)

```
    catch(Exception error)
    {
        Console.WriteLine("Message: " +
            error.ToString());
        continue;
    }

  } while (M < 1);
```

overflow Exception

```
while (true)  //infinite loop for illustration purposes only
    {
       try
       {
          Console.Write("Take product in checked mode.");
          N = checked(N * 2 );
          Console.WriteLine("The next product is  " + N);
       }
       catch (OverflowException overflowException)
       {
          Console.WriteLine(overflowException.ToString());
          break;
       }
    }
```

Now run the program

1. What happens with bad data in each case?
2. What happens if checked is removed in the segment on overflow exceptions?
3. Carefully comment out the try and the catch blocks and see what happens. Of course, in this part you need to leave the block of code that resides in the try block.

File IO

"C# views each file as a sequential *stream* of bytes. Each file ends either with an *end-of-file* marker or at a specific byte number that is recorded in a system-maintained administrative data structure." (Textbook – page 759)

- To perform file processing in C#, namespace **System.IO** must be referenced.
- There are two classes that we will concentrate on for doing I/O.
    - StreamWriter
    - StreamReader

  You can probably guess which does what.

The following code segment prepares a file for output.

```
StreamWriter outStream = null;
string outFileName = "FILEIO_1.txt";
try
{
   outStream = new StreamWriter(outFileName);
}
catch(Exception e)
{
   Console.WriteLine("Can't open file {0} ",outFileName);
   Console.WriteLine("The reason is: {0}",e.ToString());
   Environment.Exit(1);
}
```

Questions

- What is the name of the object for doing output?
- What is the name of the file?
- Why is the attempt to open the file placed within a try block? What might cause it to fail?

Once open what does the following code segment do?

```
int i,j;
for(i=1; i<= 10; i++)
{
    for(j=1; j<=10; j++)
        outStream.Write((j+i) + " ");
    outStream.WriteLine();
}

outStream.Close();
```

Question

What are the primary differences and similarities between Console.Write, Console.WriteLine, outStream.Write, and outStream.WriteLine?

Now we will run the program.

It’s name is FileIO_1.cs.

Download it and run it, but before you run it, do a dir *.txt.   After you run it do a dir *.txt

Open the new file in Notepad. Is it what we expected?

Now for some file reading

```
try
{
    inStream = new StreamReader(inFileName);
}
catch (Exception e)
{
    Console.WriteLine("Can't open file {0} ", inFileName);
    Console.WriteLine("The reason is: {0}", e.ToString());
    Environment.Exit(1);
}
```

```
string str;

str = inStream.ReadLine();
while(str != null)
{
    Console.WriteLine(str);
    str = inStream.ReadLine();
}
```

An alternate way to read the file. It currently is commented out in the program.

```
//    while ((i = inStream.Read()) != -1)
//    {
//            Console.Write((char)i);
//    }
```