

## File Input/Output (I/O) and Exceptions

### Summing Up

You have learned how to control your programs' structure using conditional statements like `if/else` and `switch`, as well as loop structures like `while` and `for`. You have also learned how to store large sets of data into arrays. However, the interaction in your programs so far has been limited to keyboard input.

### Coming Up

This chapter presents a method for reading input from files and writing output to files. You will also see how very large objects can be written to a file before a program terminates, so they can be read back in to the most recent state when the program starts up later. After studying this chapter, you will be able to:

- use some of C#'s input/output classes to read and write files.
- save data to a file for later use.
- understand C#'s exception-handling mechanism.
- learn to try, throw, catch, and avoid exceptions.
- save and restore objects using serialization and deserialization (optional).

## 8.1 Files, Streams, Readers, and Writers

When a program is running, it stores and updates data—such as variables, objects, and so on—in memory. Memory is sometimes referred to as *primary storage* in a computer because it is often the first place data is stored. However, primary storage in memory is not permanent. When a program terminates, the memory associated with that program is cleared. The data is lost.

You will often need to save data after a program ends, or to read data from a source outside the program. Usually this data comes from a disk storage device such as a hard disk, shared network drive, CD-ROM, or floppy disk. These kinds of storage devices are called *secondary storage*. They are more permanent than primary storage in memory, because if a program saves data to a disk, the data remains after the program terminates.

Data on secondary storage devices is organized into files. A *file* is a named collection of bytes. The file ends with a special byte marker called an *end-of-file marker* (often called *EOF* for short), so that programs can tell how big the file is, and when to stop reading data from it. A *directory* (also sometimes called a *folder*) is a collection of files. A directory can also contain other directories. The following figure shows a file that contains  $n$  bytes of data:

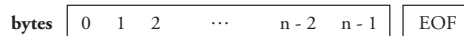


Figure 8.1: A file

You have worked with files before. Each C# program you have written was a file with a name ending in `.cs`. In this chapter, you write programs that save their data to be used after the program ends. Two common methods are to write the data into a file on the computer's hard disk, or on a network drive. The next time it runs, the program can read the data back from the same file.

C# has libraries of classes for reading and writing data to files. However, in C# the focus is not on the source of the data (the file), but instead on the transmission of data from one place to another. Input and output have been generalized in C#. This means that the methods and classes used to read data from the screen or from a file on a hard disk are the same as the methods and classes used to read data from other sources, such as the Internet or a CD-ROM.

In C#, data channels from one point to another are represented by objects called streams. A *stream* is a transmission channel through which a sequential group of bytes can be read or written from one place to another. Usually, streams are used in conjunction with *readers* and *writers*. These are unidirectional data transfer objects that can send data to, or read data from, a stream.

To read a file in a C# program, you must:

1. create a stream object that represents a connection to that file
2. get a reader for that stream
3. read the bytes in order from the reader

Fortunately, you can open a stream to almost any data source, such a connection over a local network, a Web page, or even directly from memory. This lets you read data from almost anywhere.

The C# libraries for reading and writing data using streams are found in the `System.IO` namespace. To use streams, readers, and writers in your code, you must put `using System.IO;` at the top of your program.

## Reading Files with `StreamReader`

Formally, the correct way to read a file is to construct a stream that represents a connection to that input source, such as a file. Then you connect the stream to a reader that reads data from the stream. However, C# also makes it easier to read a file without these multiple steps. The easier way is to construct a `StreamReader` object by passing a `string` for the file you want to read. Here is the general form for doing this:

---

General Form: Constructing a `StreamReader` to read from a file

```
StreamReader variable-name = new StreamReader(file-name);
```

**Examples:**

```
StreamReader reader = new StreamReader("input.txt");
```

```
StreamReader reader2 = new StreamReader("C:\\data\\taxes.dat");
```

**Notes:**

If *file-name* is not found, trying to construct the stream reader will throw a `FileNotFoundException`, which will terminate the program.

---

`StreamReader` objects have many useful methods. Here is a partial list:

**`StreamReader` constructor**

```
public StreamReader(string filename)
```

Constructs a new stream reader to read data from the specified file.

**`StreamReader` object (instance) methods**

```
public void Close()
```

Shuts down the reader and stops reading data from the stream.

```
public int Peek()
```

Looks ahead to the current character of data in the stream and returns it as an `int`, without advancing the reader; returns `-1` if end of stream has been reached.

```
public int Read()
```

Reads one byte of data from the stream and returns it as an int; returns `-1` if end of stream has been reached.

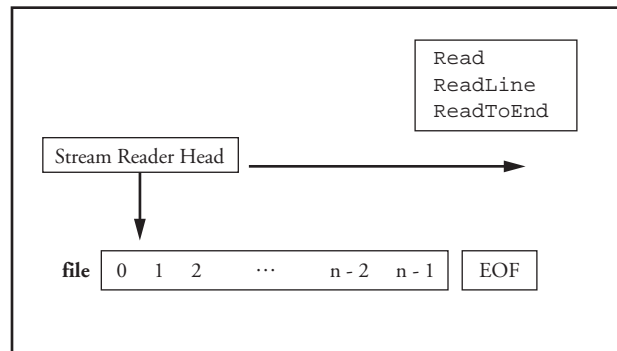
```
public string ReadLine()
```

Reads many bytes of data as characters until an end-of-line (`\n`) character is reached, and returns the characters as a string; returns null if end of stream has been reached.

```
public string ReadToEnd()
```

Reads characters from the reader's current position to the end of the file or data source, and returns the characters as a string; returns null if end of stream has been reached.

To visualize how a reader reads data from a file, picture it as having a reading head that points to a particular place in the file. As you read bytes or lines from the file, the reader advances its reading head through the file. Peeking ahead one byte does not move the reading head. The head cannot go past the end-of-file marker at the end of the stream's data.



**Figure 8.2: Stream reader diagram**

Since files can contain arbitrarily large amounts of data, programs usually read files by using an indeterminate `while` loop that grabs the next character or line. After data is read, the loop's condition tests if the file has more data; if so, it then reads the next piece of data. As we will see, it is important to see if the file has more data or not. This can be done in several ways. One way is to use the `Peek` method to see if the next byte in the file is `-1`; if so, the reader has reached the end of the file, and reading should stop. Another way is to call `ReadLine` or `ReadToEnd` and see if they return a `null` string. If so, the end of the file has been reached.

## Chapter 8 File Input/Output (I/O) and Exceptions

The following program reads a file `input.txt` and writes its contents to the screen. For the program to work, there must be a file named `input.txt` in the same folder as the folder from which the program is run.

### Code Sample: Class ReadEntireFile

```
1 using System;
2 using System.IO;
3
4 class ReadEntireFile
5 {
6     static void Main()
7     {
8         // open a stream reader to read the file
9         StreamReader reader = new StreamReader("input.txt");
10
11        // keep track of line numbers
12        int lineNum = 1;
13
14        // read each line from the file and print it with its number
15        while (reader.Peek() != -1)
16        {
17            string line = reader.ReadLine();
18            Console.WriteLine("Line {0}: {1}", lineNum, line);
19            lineNum++;
20        }
21
22        // done reading; close the file
23        reader.Close();
24    }
25 }
```

### Input file "input.txt"

```
here are two lines
of text
here is          another

that last one was a blank line

so was that one
this is the last line
```

## Output

```
Line 1: here are two lines
Line 2: of text
Line 3: here is          another
Line 4:
Line 5: that last one was a blank line
Line 6:
Line 7: so was that one
Line 8: this is the last line
```

---

## Self-Check

- 8-1 Name two sources of data for which a C# stream can be constructed to read the data.
- 8-2 What does it mean when a call to `Peek` on a stream reader returns `-1`?
- 8-3 What could you assume if a stream reader returned `-1` from its `Peek` method the very first time `Peek` was called, before any information was even read from the file?

## Specifying Path Names

The previous example used `input.txt` as the file name to read. No folder or directory information was given about where `input.txt` existed on the disk. When no folder name is given, the input file `input.txt` must be in the same folder as the executable program.

When the input file is in a different folder than the executable program, you must put a folder name before the file name, such as `myfiles\inputdata\input.txt`. If this string were passed to the `StreamReader`, it would look for a file named `input.txt` in the `myfiles\inputdata` subfolder of the folder from which the program was run. These kinds of file name specifications, which may specify no folder at all, or which may specify a folder that is a subfolder of the current directory, are called *relative paths*.

Beware of the common mistake of the leading backslash when specifying relative paths. A relative path of `myfiles\inputdata\input.txt` is very different from a relative path of `\myfiles\inputdata\input.txt`. (note the leading backslash). The former refers to the file `input.txt` in the `myfiles\inputdata` subfolder of the current folder; if the program was run from the folder `C:\school\programs`, the entire path would be `C:\school\programs\myfiles\inputdata\input.txt`. The latter refers to the file `input.txt` in the `\myfiles\inputdata` folder on the current drive (regardless of the current working folder); if

the program was run from the same folder `C:\school\programs`, the entire path would be `C:\myfiles\inputdata\input.txt`. In almost all cases, the intention is *not* to use the leading backslash; be cautious when specifying paths to avoid this very common and confusing mistake.

It is also legal to specify a fully qualified path to the file name. A fully qualified path, or *absolute path*, includes the entire path to the file, including the drive letter (if applicable) and all folders and subfolders that the file is in. An example of an absolute path would be

```
C:\programs\myfiles\inputdata\input.txt.
```

The advantage of specifying an absolute path is that the input file does not have to be in the same folder as the executable program, or in a subfolder of that folder. The disadvantage is that the file location is fixed. If the file is moved to a different folder, the absolute path string is out of sync and must be changed. For simplicity and versatility, all examples in this chapter use relative paths for their input and output files.

Since folder names are separated by `\` characters on Windows systems, and `\` specifies special escape sequences in C#, it can be inconvenient to express file paths as strings. For example, to specify the above absolute path, the following string would be needed:

```
"C:\\programs\\myfiles\\inputdata\\input.txt"
```

The C# compiler interprets each double backslash (`\\`) as being a single backslash. Without the double backslashes, the string would not compile correctly.

However, there is an easier way to specify paths. To do so, place the `@` character before the path name. Instead of the above string with double-backslashes, you can write a string like:

```
@"C:\programs\myfiles\inputdata\input.txt"
```

Using this type of string tells the C# compiler not to interpret the backslashes as control characters. Such a string is called an *uninterpreted string*, because special character combinations such as `\n` and `\\` are not interpreted as special characters. Using uninterpreted strings lets you write more convenient and readable code for constructing stream readers to read long path strings, such as:

```
StreamReader reader = new StreamReader(@"H:\home\programs\data.dat");
```

You must be careful when you place a quotation mark character in an uninterpreted string. Using `\` will not work, because the `\` will no longer be interpreted as a special character. Because of this, the first quotation mark encountered will be interpreted as making the end of the string. Instead, in an uninterpreted string, always use two quotation mark characters back-to-back (`""`) to indicate a quotation mark character:

```
string str = @"Bob said, "Hello there!" to me.";
Console.WriteLine(str);
```

---

Output

```
Bob said, "Hello there!" to me.
```

---

## Reading and Processing a File of Data

Often a file represents a list of entries of data. The data entries can be read one-by-one in your program and processed individually. All the ways that you learned in previous chapters for reading and parsing input from the console will also work here. The only difference is that with files, the contents are usually fixed before your program runs, not generated on-the-fly by users at the keyboard. With keyboard input, you can use the Prompt and Input pattern to prompt the user for input in a particular format. But with file input, you cannot do this. This is not a large problem; if you know the format of the data in the file ahead of time, you can write code to correctly read that data and process it.

Consider the following `Accumulator` class, which maintains a running sum and average of a group of numbers. When a new number is added to the `Accumulator`, it increments its count of how many numbers have been added, and it adds the number to its running sum. This allows the program to compute the sum and average.

---

Code Sample: Class Accumulator

```
1 using System;
2
3 // Adds numbers and reports the count, sum, and average.
4 class Accumulator
5 {
6     private double sum;    // fields
7     private int count;
8
9     public Accumulator()
10    {
11        sum = 0.0;          // initialize fields
12        count = 0;
13    }
14
15    // The sum of all the numbers accumulated so far.
16    public double Sum
17    {
18        get { return sum; }
19    }
20 }
```

## Chapter 8 File Input/Output (I/O) and Exceptions

```
19  }
20
21  // The count of how many numbers have been accumulated.
22  public int Count
23  {
24      get { return count; }
25  }
26
27  // Determine average; 0.0 when there are no numbers
28  public double Average
29  {
30      get
31      {
32          if (count > 0)
33              return sum / count;
34          else // avoid division by zero
35              return 0.0;
36      }
37  }
38
39  // Add one number to this accumulator.
40  public void Add(double number)
41  {
42      sum += number;
43      count++;
44  }
45 }
```

Now that there is an `Accumulator` to work with, imagine a file full of numbers to read, named `numbers.txt`. A C# program can be written that reads each number from this file and then turns it over to the `Accumulator` to add to its running sum. After each number is added, stats can be written on the screen about the numbers that have been read, and about the accumulator data.

### Code Sample: Class UseAccumulator

```
1 using System;
2 using System.IO;
3
4 class UseAccumulator
5 {
6     static void Main()
7     {
8         Accumulator accum = new Accumulator();
9         StreamReader reader = new StreamReader("numbers.txt");
```

```

10
11 // write an output header
12 Console.WriteLine("{0,10}{1,10}{2,10}{3,10}",
13                   "COUNT", "NUMBER", "SUM", "AVERAGE");
14 Console.WriteLine(new string('=', 40));
15
16 // read each line of input and write a summary line of output
17 while (reader.Peek() != -1)
18 {
19     double number = double.Parse(reader.ReadLine());
20     accum.Add(number);
21     Console.WriteLine("{0,10}{1,10:F1}{2,10:F1}{3,10:F1}",
22                       accum.Count, number, accum.Sum,
23                       accum.Average);
24 }
25
26 reader.Close();
27 }
28 }

```

input file "numbers.txt"

```

12.0
23.2
16.7
55.3
34.0
27.1
63.4
6.0
75.9
41.4
83.3
19.0

```

Output

COUNT	NUMBER	SUM	AVERAGE
1	12.0	12.0	12.0
2	23.2	35.2	17.6
3	16.7	51.9	17.3
4	55.3	107.2	26.8

5	34.0	141.2	28.2
6	27.1	168.3	28.1
7	63.4	231.7	33.1
8	6.0	237.7	29.7
9	75.9	313.6	34.8
10	41.4	355.0	35.5
11	83.3	438.3	39.8
12	19.0	457.3	38.1

## Self-Check

8-4 If a C# program is running from the `C:\Programs\Test` folder, give an absolute and relative (if possible) path string to represent each of the following:

- (a) `C:\Programs\Test\MyCode.cs`
- (b) `C:\Programs\Test\input\debug\data.txt`
- (d) `D:\WINNT\SYSTEM32\MSHTML.DLL`

8-5 Write a complete C# program that reads the file `input.dat` and grabs the first letter of each line in the file. The program must concatenate the letters together to form a combined string. For example, if `input.dat` contains this text:

```
students
have
always
respected
programmers
```

The output from the program is:

```
Message: sharp
```

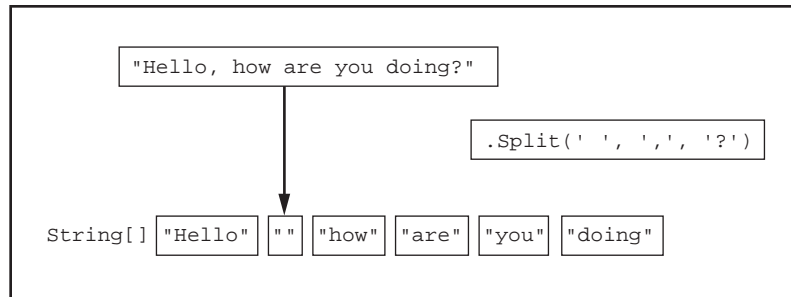
## Complex Input and String Splitting

The preceding example read an input file that contained numbers, one per line. The format of the file made it relatively easy to read it into the program. However, sometimes you must deal with more complex input, such as a file that contains multiple pieces of data on each line. When this is the case, the `Split` method of a `string` object can be very useful. `Split` is a method that tokenizes a string; to *tokenize* means to break it into separate pieces or tokens. `Split` breaks a string into smaller pieces, based on the criteria you give it. It returns the pieces as an array of `strings`. When calling `Split`, you must pass a group of `char` characters that represent the markers that tell `Split` where to chop the string up. These markers are called the *delimiters*.

One common use of `Split` is to split a line into words, based on whitespace. The figure below shows how the following string is split into an array of smaller strings:

```
"Hello, how are you doing?"
```

Sending a `Split` message to this string, and passing in as delimiters ' ' (blank space character), ',' (comma), and '?' (question mark) causes these three characters to be seen as places to chop the string. The delimiters are not included in the broken smaller strings returned by `Split`. Consecutive occurrences of the same delimiter are treated as one delimiter, but consecutive occurrences of different delimiters are treated separately. So, the ", " between `Hello` and `how` is considered two delimiters, with an empty string of "" between them.



**Figure 8.3: Splitting a string**

```

string str = "Hello, how are you doing?"
string[] tokens = str.Split(' ', ',', '?');
foreach (string token in tokens)
    Console.WriteLine(token);
  
```

#### Output

```

Hello

how
are
you
doing
  
```

String splitting is very useful when you process complex input files. For example, consider an input file named `students.txt` in the following format. The data in the file represents a list of students, listing each student's name, ID, and GPA.

## Chapter 8 File Input/Output (I/O) and Exceptions

Jones, Cindy	5193	3.75
Spattery, Thomas	3267	2.47
Richards, Marcy	2359	2.98
Millor, Jesse	0976	3.16

It would be very tedious to parse such a file using previous `string` methods, like `IndexOf` and `Substring`, to extract the pieces of information from each line. However, using `Split` with the right delimiters makes the parsing much easier.

To remove all unwanted characters from the string, it is important to specify the proper set of delimiters. In the sample data shown above, the regions containing the student IDs and GPAs line up because there is a tab character after the students' names. So, using the tab character as a delimiter (`'\t'`) is a good start. You also need to specify as delimiters the standard space character (`' '`), which precedes the last names and the GPAs, and the comma (`','`), which separates the last names from the first names.

The following program reads `students.txt` and splits the lines to extract the useful tokens. Each student's information is put into a `Student` object, which is then written to the screen.

---

### Code Sample: Class Student

```
1 using System;
2 using System.IO;
3
4 // Represents a student and his/her grades and student ID.
5 class Student
6 {
7     private string name;
8     private int id;
9     private double gpa;
10
11     public Student(string init_name, int init_id, double init_gpa)
12     {
13         name = init_name; // initialize fields
14         id = init_id;
15         gpa = init_gpa;
16     }
17
18     public override string ToString()
19     { // returns string with student's name, ID, GPA
20         return string.Format("[Name={0,-16} ID={1:D4} GPA={2:F2}]",
21                               name, id, gpa);
22     }
23 }
24
25 // Reads student information from a file and creates
26 // Student objects to store it.
27 class ReadStudentsFromFile
```

```

28 {
29     static void Main()
30     {
31         StreamReader reader = new StreamReader("students.txt");
32
33         while (reader.Peek() != -1)
34         {
35             string line = reader.ReadLine();
36             string[] tokens = line.Split(' ', '\t', ',');
37
38             // now tokens array looks like this:
39             // ["Jones" "" "Cindy" "5193" "3.75"]
40
41             // build name from last name and first name
42             string name = tokens[2] + " " + tokens[0];
43
44             int id = int.Parse(tokens[3]);
45             double gpa = double.Parse(tokens[4]);
46
47             Student stu = new Student(name, id, gpa);
48             Console.WriteLine("Student: {0}", stu);
49         }
50
51         reader.Close();
52     }
53 }

```

---

#### Output

Student: [Name=Cindy Jones	ID=5193	GPA=3.75]
Student: [Name=Thomas Spattery	ID=3267	GPA=2.47]
Student: [Name=Marcy Richards	ID=2359	GPA=2.98]
Student: [Name=Jesse Millor	ID=0976	GPA=3.16]

---

## File I/O with Command-Line Arguments

As discussed in the previous chapter, it is legal to declare `Main` to have a parameter, which is an array of strings for command-line arguments.

```
static void Main(string[] args)
```

Command-line arguments (parameters entered after the program name on the command line) are particularly useful when you write a program that reads files. For example, rather than hard-coding the file names being read, as in the previous programs in this chapter, the `Main` method can

read the file names as command-line arguments to the program. To ensure that an argument is passed, you can check the `Length` property of the `args` array to ensure that it is non-zero.

Code Sample: Class FileDisplayer

```
1 // Displays a file with line numbers on each line.
2 using System;
3 using System.IO;
4
5 class FileDisplayer
6 {
7     static void Main(string[] args)
8     {
9         if (args.Length == 0)
10            Console.WriteLine("Usage: FileDisplayer [file-name]");
11        else
12        {
13            // open the file
14            string filename = args[0];
15            StreamReader reader = new StreamReader(filename);
16
17            int lineNum = 1;
18            while (reader.Peek() != -1)
19            {
20                // read a line
21                string line = reader.ReadLine();
22
23                // write the line with a line number at the start
24                Console.WriteLine("{0,3} {1}", lineNum, line);
25                lineNum++;
26            }
27        }
28    }
29 }
```

The following output is generated when this command is entered at the command line:  
`FileDisplayer FileDisplayer.cs`

Output (the line numbers are part of the output)

```
1 using System;
2 using System.IO;
3
4 class FileDisplayer
5 {
6     static void Main(string[] args)
7     {
```

```

8     if (args.Length == 0)
9         Console.WriteLine("Usage: FileDisplayer [file-name]");
10    else
11    {
12        string filename = args[0];
13        StreamReader reader = new StreamReader(filename);
14        int lineNumber = 1;
15        while (reader.Peek() != -1)
16        {
17            string line = reader.ReadLine();
18            Console.WriteLine("{0,3} {1}", lineNumber, line);
19            lineNumber++;
20        }
21    }
22 }
23 }

```

## Self-Check

- 8-6** How many elements are in each of the following arrays, given the following variable declaration?

```
string msg = "Marty $12.50 | Kate $50.00 | Julie $20.56";
```

- (a) `msg.Split(' ')`
  - (b) `msg.Split('|')`
  - (c) `msg.Split(' ', '|')`
  - (d) `msg.Split('$', '.', '|')`
- 8-7** Write a complete C# program that accepts any number of file names as arguments and writes each file's entire contents to the screen. You may assume that two arguments are passed to the program.
- 8-8** Write a complete C# program that takes an absolute file path as its argument and constructs a tree output format of the path hierarchy. If the program is run with an argument of "C:\Documents\Csharp\Program1.cs", the output would be

```

C:
|
+--Documents
|
+--Csharp
|
+--Program1.cs

```

Assume that the argument is in the proper format.

## Writing to Files with `StreamWriter`

Just as there is a `StreamReader` that can be used to read files, there is a `StreamWriter` to write data to a stream, such as a file.

General Form: Constructing a `StreamWriter` to write to a file

```
StreamWriter variable-name = new StreamWriter(file-name);
```

**Example:**

```
StreamWriter writer = new StreamWriter("output.txt");
```

**Note:**

If the file with *file-name* does not exist, a new one is created. If a file with *file-name* does exist, it is erased (or new text can be added to it, if `true` is given as a second argument in the constructor).

Here is a summary of the useful methods and properties found in a `StreamWriter`:

### **StreamWriter constructors**

---

```
public StreamWriter(string fileName)
```

Constructs a new `StreamWriter` to write data to the stream represented by the given file name. By default, overwrites any data previously in the file.

```
public StreamWriter(string fileName, bool append)
```

Constructs a new `StreamWriter` to write data to the stream represented by the given file name. If `append` is `true`, existing contents of the file are kept and new data is written after them. Otherwise, new data replaces the existing contents of the file.

### **StreamWriter object (instance) properties**

---

```
public bool AutoFlush {get; set;}
```

Represents whether this writer should commit its writes immediately to the underlying stream by calling `Flush` after each call to `Write`.

### **StreamWriter object (instance) methods**

---

```
public void Close()
```

Shuts down the reader and stops reading data from the stream.

```
public void Flush()
```

Clears any possible buffers in the writer and forces its data through the underlying output stream.

```
public void Write(string str)
Writes all characters in str to this writer's output stream.
```

To write data to a file, construct a `StreamWriter` to write to that file's stream, then call `Write` on a `StreamWriter` and pass it a `string` to write into the file. If the file does not exist, it will be created. If the file already exists and has contents, the contents will be erased and replaced by what you write into the file (unless `append` is set to `true`, which is not done in the following examples).

You might be tempted to use a `StreamWriter` much like you use `Console.WriteLine`, but there are a few differences in the syntax. First, there is no `WriteLine` in a `StreamWriter`, only `Write`. So if you want to move to the next line, you must do so manually by using a `\n` character. Second, the `Write` method in `StreamWriter` takes only a `string`, not a format string with parameters following it. If you want complex formatted output like you have used with `Console.WriteLine`, use `string.Format` to build a format string, then `Write` that string. The following short program shows an example of writing to a file named `output.txt`.

---

Code Sample: Class `WriteToFile`

```
1 // Writes some data to a file using a StreamWriter.
2 using System;
3 using System.IO;
4
5 class WriteToFile
6 {
7     static void Main()
8     {
9         StreamWriter writer = new StreamWriter("output.txt");
10        writer.Write("Here is a line of text.\n");
11        writer.Write("\n");    // need \n to move to next line
12        writer.Write("That last one was a blank line!\n");
13        writer.Write(string.Format("A format string: {0:C} {1:F2}\n",
14                                2.0, 1.234));
15        writer.Write("This is the end of the file.");
16        writer.Close();
17    }
18 }
```

---

## Output to file "output.txt"

```
Here is a line of text.  
  
That last one was a blank line!  
A format string: $2.00 1.23  
This is the end of the file.
```

It is important to remember to close the output stream by calling `Close` on the stream writer. If this is not done, there is a chance that the lines you write to the file may be lost. On some C# implementations, output is buffered. This means that it is held in a special area of memory and is not actually written out until the stream is flushed or closed. To *flush* a stream means to force all bytes through it immediately, which causes its output to be written. Closing a stream writer flushes its stream, as does calling the `Flush` method on the writer. You can also set the `AutoFlush` Boolean property to `true` to force each `Write` to be committed to the file immediately.

---

### Self-Check

- 8-9 A `StreamWriter` has only `Write`, not `WriteLine`, and its `Write` method only takes one argument: the string to write. How can you get around these differences from `Console.WriteLine` and still go to the next line, or use formatted text strings?
- 8-10 What should you always do after finishing writing to a stream writer, to ensure that the data actually gets written?
- 8-11 Write a complete C# program that accepts two file names as arguments. The program reads the first file and writes each of its lines, reversed, into the other file. If your program is run with `in.txt` and `out.txt` as its arguments, and `in.txt` has the following contents:

```
hello  
how are you
```

After the program is run, `out.txt` will contain the following contents:

```
olleh  
uoy era woh
```

## 8.2 Exceptions

When programs run, errors occur. Perhaps the user enters a string that is supposed to be a number. When it gets parsed, the `int.Parse` or `double.Parse` method discovers that the string does not represent a valid number. Or perhaps an arithmetic expression results in division by zero. Or an array subscript is out of bounds. Perhaps there is attempt to read a file from a floppy disk, but there is no disk in the floppy drive. Perhaps a file with a specific name does not exist. In C#, an error that occurs while the program is running is called an *exception*.

You have probably already seen the word “exception” several times, since it is part of the error output that is written to the screen when a program crashes. The act of an exception occurring is called the *throwing* of an exception. When an exception is thrown and nothing is done to stop or correct it, the program crashes (which you have no doubt experienced first-hand by now!).

Exceptions in C# are represented by objects with type names that are representative of the type of error. Consider what happens when a user enters a `string` input that does not represent a valid number. During the `Parse` method call, the code recognizes this exceptional event and throws a `FormatException`:

Code Sample: Class `CauseException`

```
1 // Demonstrates causing an exception when input is not a number.
2 using System;
3
4 class CauseException
5 {
6     static void Main()
7     {
8         Console.Write("Enter an integer: ");
9
10        string numString = Console.ReadLine();
11        int anInt = int.Parse(numString);
12
13        Console.WriteLine("{0} stored in number as {1}",
14                            numString, anInt);
15        Console.ReadLine();
16    }
17 }
```

---

Dialogue (when the integer is valid)

```
Enter an integer: 123  
123 stored in number as 123
```

---

---

Dialogue (when the integer is not valid)

```
Enter an integer: oops!  
An unhandled exception of type 'System.FormatException' occurred in  
mscorlib.dll  
Additional information: Input string was not in a correct format.
```

---

It is impossible to predict when a user will enter an invalid number. But the chances are very good that it will happen at one point or another. This type of problem can be dealt with in several ways. The first is the way you have dealt with exceptions so far: to ignore it. This solution is easy on the programmer, but it leaves the program not robust to errors, and more likely to crash during execution.

The next technique is to try your best to avoid errors beforehand by carefully checking conditions whenever possible. For example, you could look at each character in the string read in with the `ReadLine` method call, checking if it is a digit; this might help to ensure that the string was a valid number and could be parsed. This technique is called *exception avoidance*. Exception avoidance is important and should be used when possible, but it can be more trouble than it is worth. Can you imagine how tedious it would be to check every letter of every line read in from the keyboard, just to avoid a `FormatException` and a program crash?

Not only is exception avoidance inconvenient at times, but it is sometimes impossible. For example, when you try to open a file to read it using a `StreamReader`, the construction of the stream reader can throw a `FileNotFoundException` if the file you specify does not exist. But using the classes we have discussed so far, you cannot accurately avoid this exception ahead of time. There is no easy way to confirm that the file exists before trying to read it, using the data types discussed so far.

## Exception Handling

If exception avoidance is difficult or impossible, you can use another technique: *exception handling*. Just as the occurrence of an exception is called throwing, the handling of an exception is called *catching* the exception. To catch an exception in C#, you must enclose a group of statements in a special block called a *try block*. A `try` block is written by writing the keyword `try`, then a pair of braces with statements between them. A `try` block must be followed by a

**catch block**, which specifies what types of exception(s) you intend to handle, and what you intend to do about them. If any statement in the `try` block throws an exception for which you have a `catch` block, the code in the `catch` block is executed, and the program does not crash.

Here is the general form for exception handling in C# using `try` and `catch` blocks:

---

General Form: Exception handling

```
try
{
    code that may possibly throw an exception
}
catch (exception-type variable-name)
{
    code to handle the exception
}
```

**Example:**

```
try
{
    Console.WriteLine("Enter two numbers");
    int number1 = int.Parse(Console.ReadLine());
    double number2 = double.Parse(Console.ReadLine());
}
catch (FormatException exception)
{
    Console.WriteLine("A problem occurred with the input!");
    Console.WriteLine("Details: {0}", exception.Message);
}
```

---

The *variable-name* that declares a name for the exception variable is optional; you should only give the exception variable a name if you intend to use it in the `catch` block. A common way to use the exception variable in the `catch` block is to write it to the screen, or to print its contained error message.

The *exception-type* specifies a C# type representing some kind of exception. There are many types of exception objects in C#. Here is a partial list:

Type	Description
Exception	general errors that happen during program execution
ArgumentException	an argument passed to a method was invalid
ArithmeticException	inability to perform arithmetic, cast, or conversion
DivideByZeroException	attempt to divide an integer value by zero
FileNotFoundException	attempt to access a file that does not exist on disk
FormatException	incorrectly formatted string, number, etc.
IndexOutOfRangeException	attempt to access outside the bounds of an array
IOException	failure to perform I/O, such as to the hard disk
NullReferenceException	attempt to send a message to a null reference variable
OutOfMemoryException	not enough memory to continue running the program
OverflowException	arithmetic operation goes outside range of int type

Of the exceptions shown above, `IOException` and `FileNotFoundException` belong to the namespace `System.IO`. The rest shown belong to the `System` namespace.

Exception objects in C# contain state that represents the type of error that occurred, as well as more information about the error. Here is a partial list of useful methods and properties that you can use on an exception object. The exception object only exists inside its corresponding catch block.

#### Object (instance) properties found in exception objects

```
public string Message {get;}
```

Returns a string representation of the error that occurred.

```
public string Source {get; set;}
```

Returns a string representation of the application or object that caused the error.

#### Object (instance) methods found in exception objects

```
public override string ToString()
```

Returns a string representation of this exception, plus the list of methods that led to the error.

The following `Main` method provides an example of handling a `FormatException`. The exception handling code (in the `catch` block) executes only when the code in the `try` block throws an exception. This avoids premature program termination when the input string contains an invalid number.

## Code Sample: Class HandleException

```
1 // Demonstrates handling one exception (FormatException).
2 using System;
3
4 class HandleException
5 {
6     public static readonly double DEFAULT_VALUE = -1.0;
7
8     static void Main()
9     { // read a number
10        Console.Write("Enter an integer: ");
11        string numString = Console.ReadLine();
12        double number;
13
14        try
15        { // convert the number from string to double
16            number = double.Parse(numString);
17            Console.WriteLine("Number parsed successfully.");
18        }
19        catch (FormatException fe)
20        {
21            // Execute this code whenever Parse throws an exception
22            Console.WriteLine("Error with value {0}: {1}",
23                numString, fe.Message);
24            Console.WriteLine("Setting number to {0}", DEFAULT_VALUE);
25            number = DEFAULT_VALUE; // reset to -1
26        }
27
28        Console.WriteLine("{0} stored as {1}", numString, number);
29    }
30 }
```

## Dialogue (when the exception does not occur)

```
Enter a number: 101
Number parsed successfully.
101 stored as 101
```

---

Dialogue (when the exception is handled)

```
Enter an integer: walrus
Error with value walrus: Input string was not in a correct format.
Setting number to -1
walrus stored as -1
```

---

Instead of ignoring the possibility of exceptional events at runtime, this program now handles potential exceptions by setting the number to an arbitrary error value of `-1.0`. Notice that the code in the `catch` block only executes if the exception is actually thrown; otherwise it is skipped. Also, notice that when the exception does occur, the program stops executing code from the `try` block and immediately jumps to the `catch` block. This is why the output, "Number parsed successfully." is not shown when the exception occurs.

It is never mandatory to catch a possible exception in C#, even if you are fairly sure it will occur. This kind of exception—one that you do not need to catch if you do not so desire—is called a *runtime exception* or an *unchecked exception*. Some programming languages also have *checked exceptions*, which must be caught in order for the code to compile. Checked exceptions force the programmer to deal with error cases explicitly, but they can be constricting and annoying in cases where the programmer does not want to handle them. In C#, all exceptions are unchecked and need not be caught if you do not wish to do so.

To catch exceptions intelligently, you need to know which statements can throw which types of exceptions. Most normal statements run a small chance of generating an exception. Some of the more common possible exceptions that can be generated by normal code include dividing by zero (`DivideByZeroException`), dereferencing a null variable (`NullReferenceException`), and indexing an array improperly (`IndexOutOfRangeException`). But many method calls on various objects and classes run higher risks of generating exceptions.

Luckily, in the .NET Framework Reference Documentation, every method lists all the exceptions that it may throw. Here is an excerpt from the documentation on the `int.Parse` method:

---

#### Exceptions that can be thrown by `int.Parse`

---

<i>Exception Type</i>	<i>Condition</i>
<code>ArgumentException</code>	The argument is a null reference.
<code>FormatException</code>	The argument does not consist solely of an optional negative sign followed by a sequence of digits ranging from 0 to 9.
<code>OverflowException</code>	The argument represents a number less than <code>int.MinValue</code> or greater than <code>int.MaxValue</code> .

## Self-Check

8-12 What exceptions, if any, could be thrown by each of the following statements?

(a) `double x = 7.0 / y;`

(b) `int i = int.Parse(Console.ReadLine());`  
`int j = 7 / i;`

(c) `string[] names = new string[5];`  
`names[0] = "Austin";`  
`Console.WriteLine(names[1].ToUpper());`

(d) `StreamReader reader = new StreamReader("exception.txt");`

8-13 Accessing an array throws an `IndexOutOfRangeException` exception when there is no element at the index specified. Consider the following program:

```
using System;

class HandleArrayIndices
{
    static void Main()
    {
        string[] names = new string[] { "Jane", null, "Mary" };
        Console.Write("Index? ");

        int index = int.Parse(Console.ReadLine());
        Console.WriteLine(names[index].ToUpper());
    }
}
```

Dialogue

Index? 5

```
Unhandled Exception: System.IndexOutOfRangeException:
  Index was outside the bounds of the array.
  at HandleArrayIndices.Main()
```

Currently, the program crashes when the user types an index that is outside the bounds of the array. Rewrite the code in `Main` so that when the index is out of bounds, the `IndexOutOfRangeException` exception is caught, and “Index out of range” is output.

## Handling Multiple Exceptions

Sometimes one statement or group of statements could throw many different exceptions. When this is the case, you can write a `try` block that handles multiple types of exceptions by having multiple `catch` blocks. When an exception occurs, it jumps to the code for its corresponding catch block, if any. Here is the general form for catching multiple exceptions:

General Form: Multiple exception handling

```
try
{
    code that may possibly throw an exception
}
catch (exception-type1 variable-name1)
{
    code to handle an exception-type1
}
catch (exception-type2 variable-name2)
{
    code to handle an exception-type2
}
```

### Example:

(see class `MultipleExceptions` that follows)

Just as with an individual exception, you can handle multiple exceptions with or without giving names to the exception variables. If you give names to the variables, you must use them in some way in the `catch` block; if you do not, a compiler warning will result.

The following program catches multiple potential exceptions in the same `try` block. This is useful because the code in the `try` block could have many different types of errors, such as the user inputting an invalid number, inputting a number for an array index past the bounds of the array, or choosing indexes such that a division by zero occurs. For any exception that has a `catch` block, an error message is printed. The exception's message is also written to the screen to provide additional information.

## Code Sample: Class MultipleExceptions

```
1 // Shows handling multiple exceptions in one try block.
2 using System;
3
4 class MultipleExceptions
5 {
6     static void Main()
7     {
8         int[] numberArray = {1, 2, 3, 0, 4, 5};
9         string line = null;
10
11         try
12         {
13             // read two indices
14             Console.Write("Enter a first index: ");
15             line = Console.ReadLine();
16             int index1 = int.Parse(line);
17
18             Console.Write("Enter a second index: ");
19             line = Console.ReadLine();
20             int index2 = int.Parse(line);
21
22             // do some math on the array, and write the answer
23             int result = numberArray[index1] / numberArray[index2];
24             Console.WriteLine("Result of division is: {0}", result);
25         }
26         catch (FormatException fe)
27         { // execute this code if the string is not a number
28             Console.WriteLine("Error: Invalid number entered: {0}",
29                 line);
30             Console.WriteLine(fe.Message);
31         }
32         catch (IndexOutOfRangeException ioore)
33         { // execute this code if the numberArray[index] is bad
34             Console.WriteLine("Error: Invalid index.");
35             Console.WriteLine(ioore.Message);
36         }
37         catch (DivideByZeroException dbze)
38         { // execute this code if numberArray[3] is the denominator
39             Console.WriteLine("Error: division by zero!");
40             Console.WriteLine(dbze.Message);
41         }
42     }
43 }
```

---

Output (when no exception occurs)

```
Enter a first index: 5
Enter a second index: 1
Result of division is: 2
```

---

---

Output (when `FormatException` occurs)

```
Enter a first index: umbrella
Error: Invalid number entered: umbrella
Input string was not in a correct format.
```

---

---

Output (when `IndexOutOfRangeException` occurs)

```
Enter a first index: 15
Enter a second index: -1
Error: Invalid index.
Index was outside the bounds of the array.
```

---

---

Output (when `DivideByZeroException` occurs)

```
Enter a first index: 5
Enter a second index: 3
Error: division by zero!
Attempted to divide by zero.
```

---

Alternatively, you can catch an exception of ANY type by specifying a general class of exception to catch, rather than any specific exceptions. A `catch` block that catches type `Exception` will catch any possible exception that occurs in the `try` block.

It may seem like a great idea to always catch `Exception`, since this will make your code handle any exception that could occur. However, it is often not a good idea to use exception handling that is this general. First, different errors should be usually be handled in different ways; trying to handle them all the same way is likely to produce mediocre results. Second, catching `Exception` is also dangerous because there are usually errors that cannot be handled gracefully, and that should be allowed to crash the program.

Here is an example of catching `Exception` to handle multiple errors:

## Code Sample: Class CatchGeneralException

```
1 // Catches any exception that occurs. (bad design)
2 using System;
3
4 class CatchGeneralException
5 {
6     static void Main()
7     {
8         string[] names = new string[] {"Jane", null, "Mary"};
9         Console.Write("Index? ");
10
11        try
12        {
13            // read a number and try to print that array element
14            int index = int.Parse(Console.ReadLine());
15            Console.WriteLine(names[index].ToUpper());
16        }
17        catch (Exception ex) // catch ANY error that happens
18        {
19            Console.WriteLine("An error occurred: {0}", ex.Message);
20        }
21    }
22 }
```

## Output (when invalid number is typed (FormatException))

```
Index? Billy Bob
An error occurred: Input string was not in a correct format.
```

## Output (when names[index] is null (NullReferenceException))

```
Index? 1
An error occurred: Object reference not set to an instance of an object.
```

---

## Self-Check

- 8-14 Write code that repeatedly prompts for a number, until a valid number is entered from the keyboard. The program should re-prompt (and not crash with an exception) if an invalid number is entered. One dialog could look like this:

```

Enter a number: 101.0
Invalid number, try again
Enter a number: huh?
Invalid number, try again
Enter a number: 4.5
Valid number: 4.5

```

## Throwing Exceptions

You have now seen how to catch exceptions caused in methods that you call. It is also important to know how to create, or throw, exceptions of your own. You might ask, “Why would I want to throw an exception? That will crash the program!” The reason you would want to throw exceptions is to alert others who are using your code of errors that occur. For example, in the `BankAccount` class, when a withdrawal was made that exceeded the account balance, or when a negative withdrawal was made, the `BankAccount` simply ignored the transaction or withdrew as much as it could. A better way to handle this situation would have been to alert the caller that the amount to withdraw was invalid. The correct way to cause such an alert is by throwing an exception.

To throw an exception in C#, you must write a `throw` statement. A *throw statement* is a line of code that creates an exception object and causes the exception to occur, using the `throw` keyword. Here is the general form for throwing an exception:

---

General Form: Throwing an exception

```
throw new exception-type(“description of error”);
```

**Example:**

```
throw new Exception("Invalid data!");
throw new ArgumentException("The value must be positive");
```

---

The `Withdraw` method in `BankAccount` could be modified to throw an exception when the amount is invalid. Since `amount` is an argument, a suitable type of exception to throw would be an `ArgumentException`. Here is a possible modified `Withdraw` method:

```
public void Withdraw(double amount)
{
    if (amount < 0.0)
        throw new ArgumentException("amount is negative");
    else if (amount > balance)

```

```

        throw new ArgumentException("amount exceeds balance");
    else
        balance -= amount;    // successful
}

```

Now any code that tries to make such a withdrawal will crash the program. Here is an example:

```

BankAccount acct = new BankAccount("Starving Student", 1.00);
acct.Withdraw(10.00);

```

#### Output

```

Unhandled Exception: System.ArgumentException: amount exceeds balance
   at BankAccount.Withdraw(Double amount)
   at TestBankAccount.Main()

```

Why is this useful? For starters, it makes people who call your methods (including you!) more careful. If you know that an invalid withdrawal will crash the program, you will probably check more carefully not to do it in the first place. In other words, you are enforcing sensible coding by encouraging exception avoidance. Also, by having exceptions in your methods, you can allow the programmer to catch them and deal with them intelligently. You are allowing for intelligent exception handling. Before, there was no sophisticated way to discover if a withdrawal was valid, or successful, without doing a lot of mandatory manual checking of amounts and balances. Now, a caller can simply `try` a withdrawal and deal with its potential failure intelligently with a well-written `catch` block.

## Self-Check

- 8-15** Write a static method named `TryIt` that throws an `ArgumentException` whenever its `BankAccount` argument is `null`. When the argument is not `null`, print the `ToString` version of the argument. The following code should generate output similar to that shown:

```

TryIt(new BankAccount("Srini", 100.00));
TryIt(null);

```

## Output

```
Srini $100.00
Unhandled Exception: System.ArgumentException
```

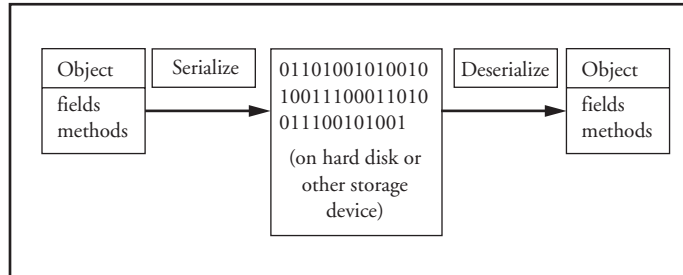
- 8-16 Modify the `BankAccount` class so that the `Deposit` and `Withdraw` methods throw an `ArgumentException` if the amount to deposit or withdraw is invalid. Invalid amounts are negative, or withdrawals that are greater than the balance of the account).

### 8.3 Advanced Topic: Persistent Objects with Serialization

The following section discusses an advanced C# topic called serialization, which allows objects to be saved and restored from the hard disk. Serialization is very powerful, but it is also somewhat complex to use in C#, because it requires a brief introduction to several topics that have not yet been covered in this textbook. Among the topics mentioned below are enumerations, attributes, typecasting, and a new .NET Framework namespace and type. Because of the number of new and advanced topics in this section, it may be considered an advanced optional topic.

You have now seen how to write text data to a file. Using the streams described above, the state of various objects can be saved to a disk by taking them apart, field by field, and writing the pieces to the disk. Although this has not been covered yet, objects could then be restored by reading in the pieces from the disk and constructing them again. But this is a lot of unnecessary work; when the objects have arrays and other complex structures, it can be very difficult. Fortunately, C# has a much more elegant solution for saving and restoring objects: serialization and deserialization. *Serialization* is the process of packaging an object and pushing it through a stream. To serialize an object, the bytes of the object are first converted into a binary format, which is then sent, in order, through the stream. Serialization allows an entire object to be saved to a file and then restored later. The object can be as simple as a string, or as complex as a collection with millions of elements.

Restoring the object later is called *deserializing* the object. This allows objects to have their state live on through multiple executions of a program. Such an object is called a *persistent object*. Persistent objects are very powerful, because they allow easy transfer of objects to files on a disk, through a network, or even to and from the Web.



**Figure 8.4: Serialization and deserialization of an object**

In C#, object serialization is accomplished through the use of a `BinaryFormatter` object. A `BinaryFormatter` object has methods to save or restore an object with a stream. `BinaryFormatter` is part of the lengthily named `System.Runtime.Serialization.Formatters.Binary` namespace.

To use the `BinaryFormatter` class, your program should contain the following `using` directive at the top:

```
using System.Runtime.Serialization.Formatters.Binary;
```

The `BinaryFormatter` has two useful methods, `Serialize` and `Deserialize`, that will respectively allow you to save and load an object from the hard disk or other storage area. Both `Serialize` and `Deserialize` require a `FileStream` object as their first argument, so that the `BinaryFormatter` knows what file to read or write. In the case of `Serialize`, there is a second argument, which specifies the object you want to save. In the case of `Deserialize`, there is no second argument; instead, the object is loaded from the file and is returned as the result of the method.

The `FileStream` constructor takes two arguments: a file name string, and a mode that explains what you wish to do with the file. For example, to construct a `FileStream` object that represents a file named `input.data`, first create a `FileStream` object like this:

```
FileStream aStream =
    new FileStream("input.data", FileMode.Create);
```

The second argument, of type `FileMode`, is a special type called an *enumeration*. Enumerations are types that only contain a fixed set of unique named instances. Enumerations are useful

to represent unique states or mutually exclusive modes. A detailed discussion of enumerations is outside the scope of this textbook; you only need to know which `FileMode` value to pass to the `BinaryFormatter` for it to do its work.

Here is the general form for saving and restoring objects to a file, using serialization and deserialization:

---

General Form: Serializing an object to a file

```
FileStream stream-name =  
    new FileStream(file-name, FileMode.Create);  
BinaryFormatter formatter-name = new BinaryFormatter();  
formatter-name.Serialize(stream-name, variable-name);
```

**Example:**

```
FileStream stream = new FileStream("account.dat",  
                                FileMode.Create);  
BinaryFormatter formatter = new BinaryFormatter();  
formatter.Serialize(stream, new BankAccount("Joe", 1.00));
```

---

---

General Form: Deserializing an object from a file

```
FileStream stream-name =  
    new FileStream(file-name, FileMode.Open);  
BinaryFormatter formatter-name = new BinaryFormatter();  
type variable-name =  
    (type) formatter-name.Deserialize(stream-name);
```

**Example:**

```
FileStream stream = new FileStream("account.dat",  
                                FileMode.Open);  
BinaryFormatter formatter = new BinaryFormatter();  
BankAccount joe = (BankAccount)formatter.Deserialize(stream);
```

---

One bizarre thing about the syntax for `Deserialize` is that the object's type must be written in parentheses before the method call, such as the `(BankAccount)` in the example above. This is because the `BinaryFormatter` can load any type of object and therefore needs information from

you about what type of object it is deserializing. This syntax in general is called a type-cast, which is not covered in detail in this textbook.

## Allowing Your Own Classes to be Serialized

There is one catch to object serialization: for it to be legal to serialize a given type of objects, the class must be tagged with a special `Serializable` attribute. An *attribute* is an identifier placed between square brackets [ and ] at the top of a class or method, which provides special information about the class. In this case, the *Serializable* attribute simply tells the .NET virtual machine that it is okay for objects of your class to be saved and loaded using serialization. Serialization involves saving an object and all of its state. This means that when you serialize an object, not only is the object itself saved, but also the values of all fields of the object. Because of this, you must make sure that all fields of any object you intend to serialize are themselves objects of serializable types. Fortunately, many of C#'s existing types are already serializable, such as `string`, `DateTime`, `Random`, and all of the basic value types like `int` and `double`. The .NET Framework Class Library Reference documentation for a particular type will state whether the type is serializable or not.

Here is the general form for adding the `Serializable` attribute to a class:

General Form: Serializable Attribute (allows objects to be saved)

```
[Serializable]
class class-name
{
    // the code for the class ...
```

### Example:

```
[Serializable]
class BankAccount
{
    string id;
```

The following program creates and saves two `Student` objects to a file using serialization, then loads them back from the file, using deserialization. This would not work if the `Student` type did not have the `[Serializable]` attribute attached.

Code Sample: Classes `Student` and `SerializeStudents`

```
1 using System;
2 using System.IO;
3 using System.Runtime.Serialization.Formatters.Binary;
```

## Chapter 8 File Input/Output (I/O) and Exceptions

```
4
5 // Represents a student and his/her grades and student ID.
6 [Serializable]
7 class Student
8 {
9     private string name;
10    private int id;
11    private double gpa;
12
13    public Student(string init_name, int init_id, double init_gpa)
14    {
15        name = init_name; // initialize fields
16        id = init_id;
17        gpa = init_gpa;
18    }
19
20    public override string ToString()
21    { // returns string with student's name, ID, GPA
22        return string.Format("[Name={0,-16} ID={1:D4} GPA={2:F2}]",
23                               name, id, gpa);
24    }
25 }
26
27 // Writes two students to disk using serialization.
28 class SerializeStudents
29 {
30     static void Main()
31     {
32         Student student1 = new Student("Billy Bob", 1234, 2.0);
33         Student student2 = new Student("Jimmy Ray", 5678, 3.1);
34
35         FileStream stream = new FileStream("students.dat",
36                                           FileMode.Create);
37         BinaryFormatter formatter = new BinaryFormatter();
38
39         // serialize them
40         Console.WriteLine("Writing Student: {0}", student1);
41         formatter.Serialize(stream, student1);
42
43         Console.WriteLine("Writing Student: {0}", student2);
44         formatter.Serialize(stream, student2);
45
46         stream.Close();
47
48         // wipe out the variables to show the deserialization works
49         student1 = null;
50         student2 = null;
```

### 8.3—Advanced Topic: Persistent Objects with Serialization

```
51
52 // read the two students back from the disk (deserialize)
53 Console.WriteLine();
54 stream = new FileStream("students.dat", FileMode.Open);
55
56 student1 = (Student)formatter.Deserialize(stream);
57 Console.WriteLine("Read a Student: {0}", student1);
58
59 student2 = (Student)formatter.Deserialize(stream);
60 Console.WriteLine("Read a Student: {0}", student2);
61
62 stream.Close();
63 }
64 }
```

#### Output

```
Writing Student: [Name=Billy Bob           ID=1234 GPA=2.00]
Writing Student: [Name=Jimmy Ray          ID=5678 GPA=3.10]

Read a Student: [Name=Billy Bob           ID=1234 GPA=2.00]
Read a Student: [Name=Jimmy Ray          ID=5678 GPA=3.10]
```

The program may not seem terribly impressive, since its output merely writes out the data for the same pair of students twice. However, it's useful because the two `Student` objects are saved in the `students.dat` file even after the program ends. This program, or another, can then be run at a later date to retrieve them.

Here is a modified version of the program:

#### Code Sample: Class DeserializeStudents

```
1 using System;
2 using System.IO;
3 using System.Runtime.Serialization.Formatters.Binary;
4
5 class DeserializeStudents
6 {
7     static void Main()
8     {
9         // read the two students saved earlier back from the disk
10        FileStream stream = new FileStream("students.dat",
11                                           FileMode.Open);
```

## Chapter 8 File Input/Output (I/O) and Exceptions

```
12     BinaryFormatter formatter = new BinaryFormatter();
13
14     Student student1 = (Student)formatter.Deserialize(stream);
15     Console.WriteLine("Read a Student: {0}", student1);
16
17     Student student2 = (Student)formatter.Deserialize(stream);
18     Console.WriteLine("Read a Student: {0}", student2);
19
20     stream.Close();
21 }
22 }
```

### Output

```
Read a Student: [Name=Billy Bob           ID=1234  GPA=2.00]
Read a Student: [Name=Jimmy Ray          ID=5678  GPA=3.10]
```

When you are deserializing objects back from a file, they must be deserialized in the same order they were serialized. For example, if you serialize a `BankAccount` and then a `Student`, you must deserialize the `BankAccount` first and then the `Student`. Trying to deserialize them in the wrong order produces an exception when trying to cast the reference to the proper type.

### Code Sample: Class SerializeObjects

```
1 using System;
2 using System.IO;
3 using System.Runtime.Serialization.Formatters.Binary;
4
5 class SerializeObjects
6 {
7     static void Main()
8     {
9         // serialize two objects
10        FileStream stream = new FileStream("objects.dat",
11                                         FileMode.Create);
12        BinaryFormatter formatter = new BinaryFormatter();
13        formatter.Serialize(stream,
14                            new Student("Billy Bob", 1234, 2.0));
15        formatter.Serialize(stream, new BankAccount("Jethro", 50.0));
16        stream.Close();
17
18        // try to deserialize the objects (in the wrong order)
19        stream = new FileStream("objects.dat", FileMode.Open);
```

```
20     BankAccount jethro = (BankAccount)formatter.Deserialize(stream);
21     Student billy = (Student)formatter.Deserialize(stream);
22 }
23 }
```

---

#### Output

```
Unhandled Exception: System.InvalidCastException: Specified cast is not
valid.
   at SerializationExceptionExample.Main()
```

---

## Self-Check

- 8-17 How could serialization be used to maintain a persistent high score list for a guessing game?
- 8-18 Write code to create and serialize two `DateTime` objects to a file named `dates.dat`.
- 8-19 Now write the code to retrieve the two `DateTime` objects from the file, using deserialization.

---

## Chapter Summary

- A file is a named collection of bytes. C# represents connections to files with objects called streams, which are manipulated using objects named readers and writers. These objects are found in the `System.IO` namespace.
- Files can be read by calling `ReadLine` on a `StreamReader` object.
- Through parsing and string splitting, you can process an input file that is in a complex format, or which has many fields of different types on each line.
- Files can be written to by calling `Write` on a `StreamWriter` object.
- Objects can be read and written through streams by using the process called serialization. This makes them persistent objects that live on after the program stops running.
- To make a type that you have created serializable, you must put the `[Serializable]` attribute atop its class header.
- Use the `Serialize` and `Deserialize` methods of a `BinaryFormatter` object (from the `System.Runtime.Serialization.Formatters.Binary` namespace) in con-

junction with a `FileStream` to save and load objects using serialization and deserialization.

- When programs run, errors often occur; in C#, such errors are called exceptions.
- Causing an error is called throwing an exception; handling the error is called catching it.
- Exceptions may often be avoided, but in other cases it is better to handle them using `try` and `catch` blocks. One `try` block may catch multiple exceptions, and may catch general categories of exceptions in order to handle many errors the same way.

---

## Key Terms

absolute path	exception	secondary storage
attribute	exception avoidance	<code>Serializable</code>
catch	exception handling	serialization
catch block	file	stream
checked exception	flush	<code>tokenize</code>
delimiter	folder	<code>throw</code>
deserialize	persistent object	<code>try</code> block
directory	primary storage	unchecked exception
end-of-file marker	reader	uninterpreted string
<code>eof</code>	relative path	writer
enumeration	runtime exception	

---

## Exercises

1. Explain the concept of a stream and why it is useful.

2. Given the following input file `input.txt`:

```
val1:val2:val3:::val4
  second line !!!
```

what would be the output from the following code?

```
StreamReader reader = new StreamReader("input.txt");
string line = reader.ReadLine();
string[] tokens = line.Split(':');
Console.WriteLine(tokens[1]);           // (a)
Console.WriteLine(tokens[4]);           // (b)
line = reader.ReadLine();
```

```

Console.WriteLine(line.Trim().ToUpper());    // (c)
tokens = line.Split(' ');
Console.WriteLine(tokens[1]);                // (d)
Console.WriteLine(reader.Peek());           // (e)

```

3. What happens when a C# program constructs a `StreamReader` object that tries to read a file that does not exist? Be specific.
4. What does the `Peek` method of a `StreamReader` return if it has reached the end of the file? What does `ReadLine` return in the same situation?
5. What new C# namespace must be included when using the input/output classes such as `StreamReader` and `StreamWriter`?
6. Are the following paths relative or absolute?
  - (a) `"foo.txt"`
  - (b) `"C:\\documents\\files\\chapter8\\foo.txt"`
  - (c) `"chapter8\\foo.txt"`
  - (d) `"files\\chapter8\\foo.txt"`
  - (e) `"H:\\foo.txt"`
7. Write the equivalent strings from the previous question, but write them as uninterpreted strings with the `@` modifier.
8. Write a method `CountLines` that takes a string argument representing a file name, counts the number of lines in that file, and returns the number of lines as an `int`. Blank lines should also be counted. Your method should work with the following test code:

```

using System;
class Lines
{
    static void Main()
    {
        int lines1 = CountLines("file1.txt");
        int lines2 = CountLines("file2.txt");
        Console.WriteLine("# of lines in file 1 = {0}", lines1);
        Console.WriteLine("# of lines in file 2 = {0}", lines2);
    }
}

```

9. Write a method `CountWords`, similar to the `CountLines` method from the previous question, but which counts the number of words in the file whose name is passed to it. A word is defined as any nonempty sequence of characters separated by whitespace (the space character, the tab character, or the new line character).
10. Write class `FileSwapper` with method `public void Swap(string file1, string file2)` that will swap the contents of two text files. That is, after the swap, `file1` will contain what used to be the contents of `file2`, and `file2` will contain what used to be the contents of `file1`.
11. Assume that the string method `split` did not exist. Write your own `split` method that takes a `string` to be split, and `char` characters for the delimiters, and that returns an array of strings representing the same results that `String.Split` would have returned. The delimiters should not be part of the result array. For example, calling your method with `Split("hello, are you there,, Suzy?", ',')` should return the following array of strings:  

```
 {"hello", " are you there", "", " Suzy?"}
```

Perhaps this goes without saying, but do not use `String.Split` to write your solution.

12. Write a program that takes two file names as command-line arguments. The program should read the first file, and then write each line reversed into the second file. For example, if your program is named `ReverseIt` and is run with the following command line:

```
ReverseIt file1.txt file2.txt
```

and if the contents of `file1.txt` are the following:

```
this is a line
here is more text
1 2 3
```

```
last text here
```

after the program runs, `file2.txt` should have the following contents:

```
enil a si siht
txet erom si ereh
3 2 1
```

```
ereh txet tsal
```

If the first argument `file` is not found, or if not enough arguments are passed, print an error message.

13. Write a program named `Compare` that takes two file names as command-line arguments and writes "Same" to the console if the two files have the same contents, or "Files differ on line N:" plus each file's line N if the files have a line that does not match. Only print the first non-matching line. For example:

```
Compare readme.txt readme.txt
Same
```

```
Compare readme.txt readme2.txt
Files differ on line 4:
readme.txt: Four score and seven years ago
readme2.txt: Fore scour and yeven sears ago
```

If either file is not found, or if not enough arguments are passed, print an error message.

14. Name all the possible exceptions that could happen in the following code:

```
int num = int.Parse(Console.ReadLine());
string[] array = new string[3];
array[1] = array[0].Substring(0, 2) + num;
array[2] = "" + (num / 10);
```

---

## Programming Tips

1. Do not try to read past the end of a file; avoid this by using `Peek` to make sure that data is still remaining.

The `Peek` method returns `-1` when there is no more data to read, so use it as the test for your loop that reads the file. This way, you can avoid exceptions that would be thrown when trying to read past the end of a file.

2. Use relative path names if possible; they are less prone to errors.

If your program is in the folder `C:\My Documents\homework5`, and you want to read the file `C:\My Documents\homework5\temp\file.txt`, ask for a reader that reads `"temp\\file.txt"` rather than `"C:\\My Documents\\homework5\\temp\\file.txt"`. This way, if your program and its temp folder ever move to a different folder, it will still behave correctly.

**3. String splitting works best when the tokens in the string are not separated by multiple delimiters.**

The following example input has fields separated by two tabs each. Notice that you cannot distinguish the tab characters from space characters on the screen or on this printed page. If you split a string on the tab character `'\t'`, and your file has input such as this:

```
Smith      Jane          $100.00      5678
```

the array returned from `Split` would contain these elements:

```
{"Smith", "", "Jane", "", "$100.00", "", "5678"}
```

If possible, only split strings that are delimited by nonconsecutive occurrences of your delimiters, or else be very certain you know the input format, so that it will be split correctly.

**4. Try to avoid exceptions first. Then, if that is not possible, handle them intelligently.**

For example, your program will throw an exception when you divide an integer by zero. You could handle this by catching the exception. However, a better solution is to do your best to avoid the exception in the first place by checking the value of numbers before dividing by them. If possible, avoiding exceptions is preferable to trying to catch them.

**5. Print verbose and informative error messages from exceptions that occur.**

When a program crashes, it is frustrating for the user. It can be even more frustrating for the programmer to debug the code without good information about what went wrong, and how to fix it. Printing exceptions' error messages (with the `Message` property in the exception object), along with your own message, is a good start toward this goal.

**6. Deserialize objects in the same order that they were serialized.**

If two objects are serialized, they must be deserialized in the same order. This is especially true of objects of different types, because the code will throw an exception if you try to typecast an object into the wrong type.

**7. Add the `[Serializable]` attribute to any class that you want to serialize.**

Most of the C# types covered in this textbook are serializable: `string`, `int`, `double`, `char`, `DateTime`, `BankAccount`, and other types can safely be saved to files. However, if you write your own types and intend to save them to disk, make sure to include the `[Serializable]` attribute just above the class's header. If the attribute is omitted, an exception will occur when the your program tries to serialize the object.

---

## Programming Projects

### 8A Above and Below

Write a C# program (the `Main` method only) that creates an undetermined number of `BankAccount` objects and stores them into an array. The input should come from an input file (see the `StreamReader` class) which contains the initial balance and the ID for each object, as shown below (use `Console.ReadLine` and the string `IndexOf` method).

Use the following input file:

```
53.45 Solley
999.99 Kirsten
8790.56 Pantone
0.00 Brendle
1555.76 Kentish
```

After initializing the array and the number of `BankAccount` objects, display every `BankAccount` that has a balance greater than 1,000.00. Then display every `BankAccount` that has a balance less than 500.00.

Your output should look like this if you use the `ToString` method of `BankAccount` when the program runs in a country that uses American dollars:

```
Accounts with balances > 1000.00
Pantone $8,790.56
Kentish $1,555.76
```

```
Accounts with balances < 500.00
Solley $53.45
Brendle $0.00
```

### 8B Frequency

Write a C# program that reads integers from a file and reports the frequency of each integer. As shown in the dialogue below, your program should ask the user for the input file name, and should then print a table of the values in the file, plus how many times each value occurred. Note that the highest numbers should appear first. The input file only has numbers that are in the range of 0 through 100 inclusive. Use this fact to your advantage. You may assume that there are never more than 100 numbers in the file. Use the `StreamReader` class to read from the disk file.

The File test.dat

```
75
60
75
100
60
75
90
85
90
```

Dialogue

```
Enter file name: test.dat
100: 1
90: 2
85: 1
75: 2
60: 2
```

## 8C ReadAmount

Write a static method named `ReadAmount` that reads input from the keyboard until the user enters a valid currency amount, and then returns that amount. Also give the user a chance to cancel the operation, if so desired, by entering *quit*, in which case the method must return 0. There are three things that your method must check for and report as invalid input:

1. A string that does not successfully parse to a valid number, such as "1x.45".
2. A string that parses to a number with more than 2 decimal places (you will need to write a formula).
3. A string that parses to a negative number, or is 0.0.

The following code should compile and show the valid currency amount (or 0 if the user quits)

```
// This code assumes ReadAmount is a static method in the same class
double number = ReadAmount("Enter a number: ");
Console.WriteLine("Valid number = {0}", number);
```

---

Output when the user enters four incorrect currency amounts

```
Enter a number: badNumber
'badNumber' is an invalid currency amount. Enter 'QUIT' or try again.
Enter a number: 1.233
'1.233' must have 2 or fewer decimals. Enter 'QUIT' or try again.
Enter a number: -1
'-1' must be positive. Enter 'QUIT' or try again.
Enter a number: 0
'0' must be positive. Enter 'QUIT' or try again.
Enter a number: 123.45
Valid number = 123.45
```

---

## 8D Grep

Write a program named `Grep` that takes a file name as its first command-line argument, and a string to search for as its second command-line argument. The program should then print the lines of the file that contain that string, preceded by the line numbers and a colon. If no lines match the string, no output is printed.

For example, if your program was run with `Grep gettysburg.txt` on (searching for the string `on` in the file below), your output would be as shown. The string `on` can occur anywhere in the line; it does not have to be a separate word.

---

Input file `gettysburg.txt`

```
Fourscore and seven years ago
our fathers brought forth on this
continent a new nation, conceived
in liberty and dedicated to the
proposition that all men are created equal.
Now we are engaged in a great civil war,
testing whether that nation or any nation so
conceived and so dedicated can long endure.
```

---

---

Output

```
2: our fathers brought forth on this
3: continent a new nation, conceived
5: proposition that all men are created equal.
7: testing whether that nation or any nation so
8: conceived and so dedicated can long endure.
```

---

## 8E Doubler

Write a program named `Doubler` that takes a file name as a command-line argument, and in the output doubles every non-whitespace character in the file, and also doubles each line in the file. For example, if your program is run with the command line `Doubler test.txt`, and the file `test.txt` contains the following contents:

```
testing
1 2 3
```

```
more data
```

the new contents of `test.txt` after running `Doubler` would be:

```
tteessttiinngg
tteessttiinngg
11 22 33
11 22 33
```

```
mmoorree ddaattaa
mmoorree ddaattaa
```

If not enough arguments are passed, or the file is not found, print an error message. (Note that you will have to catch an exception to do this.) Note that your program must write the doubled text back into the same file; it destroys the file's original contents to read and write the same file at the same time. The two operations must be performed separately, without overlapping.