Top of Form



**Introduction to C# Programming: Lesson 11**

**Chapter 1**

**Introduction**

Have you ever been running a program, either on the Internet or at home, and the program displayed a message box telling you that it encountered an error and needed to shut down? This has probably even happened to you when you are testing the programs that you have written for this course. This is usually quite frustrating, because even though the computer seems to have an idea of what the problem is, the program still crashes and you lose all of the information that you have already entered. This lesson will teach you about what these problems are and what you can do to keep them from crashing your program.

In addition, you will learn how to use files with your C# programs. Files can be used as a source for input, as well as a way to store results of your programs. This lesson will show you how to write code that will take advantage of the capabilities of files.  
  
  
  
  
**Chapter 2**

**Exception Handling**

An *exception* is an indication that some problem has occurred while the program is running. The term *exception* comes from the idea that the program usually runs without problems. However, the problem that was encountered was "an exception to the rule." Although there are many possible exceptions, some of the more common ones are:

* The user entered invalid data.
* The program is attempting to divide by zero.
* The program is attempting to access an element of an array that does not exist.

Just in case you have not seen what happens when an exception occurs, try typing in the following program:

using System;

public class GetInt

{

public static void Main()

{

String strNum;

int number;

Console.Out.Write("Enter a whole number: ");

strNum = Console.ReadLine();

number = Convert.ToInt32(strNum);

Console.Out.WriteLine("\n\nYour number ¬

was " + number);

}

}

Compile and run this code and you will find that the program runs just fine when you type in a whole number at the prompt. But run the program again, and this time type in something other than a whole number. For example, try typing *12.5* and press RETURN. When you do this, the program should stop running and a dialog box should pop up. If you read the text inside the dialog box, it should say that a SystemFormatException occurred in your program. This dialog box then wants to know if you want to run the debugger. Be sure to click **No**. Doing this will close the dialog box and stop your program.

As you can see, if the user types a whole number when expected, nothing happens. However, when the user does not enter the correct type of value, an exception occurs and the program crashes. Depending on what kind of application you are writing, your program crashing could be life threatening or just a minor inconvenience. Either way, good programmers do not let their programs terminate abruptly like this. And since you are learning how to be good programmers, I want to be sure you know how to handle such problems.

When writing your program, if there is a possibility that your code may generate an exception, you will put that code inside what is called a *try-block*. The syntax of a try-block is as follows:

*try  
{  
< statements that might create an exception>  
}*

The idea is that you are telling the computer to attempt, or try, to execute the following code. If everything goes as planned, then there is no need to do anything different. However, we know there is a possibility that an exception can occur or, in object-oriented terms, an exception is *thrown*. If this does happen, then we must *catch* the exception. This will be done is what is called a *catch-block*. The syntax of the catch-block is as follows:

*catch ( < name of exception type> < exception variable name>)  
{  
< statements to be run when an exception is thrown>  
}*

Unfortunately, the syntax of the catch-block is a little more complicated than the try-block, but I assure you that you can figure this one out. Actually, the only thing that may be a little confusing about this is the terms inside the parentheses after the keyword *catch*. If you look closely at these terms, it should remind you of the parameters inside a method header. That is because we are doing just about the same thing here as we do in a method header. You see, an exception is an object. Therefore, when an exception is thrown, we can take this exception and store it in a variable. Before you see what can be done with this variable, let's fix our program so that the exception is caught.

using System;

public class HandleGetInt

{

public static void Main()

{

String strNum;

int number;

Console.Out.Write("Enter a whole number: ");

strNum = Console.ReadLine();

try

{

number = Convert.ToInt32(strNum);

}

catch (Exception e)

{

Console.Out.WriteLine("Invalid ¬

whole number, 1 is used.");

number = 1; }

Console.Out.WriteLine("\n\nYour number ¬

was " + number);

}

}

Notice how I put only the statement that attempts to convert the string into an int into the try-block. That is because there is no way that the other statements can generate an exception. Also notice that the catch-block is attempting to catch the exception and store it in a variable named *e*.

Type this code in and compile it. You will see that the compiler gives you a warning saying that a variable *e* was declared but never used. Recall that *e* is the exception variable, and the compiler is correct: Our code does not do anything with *e*. Since this is only a warning by the compiler, it is OK to ignore it and run the program. When you run the program and type in a whole number, you will notice that the program runs just as it did before. However, run the program again and, this time, type in your name at the prompt. Since this causes the program to throw an exception, the code inside the catch-block is run and a message is displayed on the screen indicating that invalid data was entered. The code sets *number* equal to one, and the program continues with the code below the catch-block.

**Chapter 3**

**More About Exceptions**

You may be wondering what the point of catching the exception object is. Since this is an object, it means that there are some methods associated with it. One method that can be of some use to us is the *ToString()* method. This method will return a description of the object in the form of a string that can be displayed on the screen. To see an example of this, consider the following example that throws an exception when attempting to divide by zero:

using System;

public class DivideByZero

{

public static void Main()

{

int numerator = 10;

int denominator = 0;

int result;

try

{

result = numerator / denominator;

}

catch (Exception e)

{

Console.Out.WriteLine(e.ToString());

}

}

}

Compile and run this code and you will see the name of the exception that occurred (*System.DivideByZeroException*), a message (*Attempted to divide by zero*) and the location of the problem (*at DivideByZero.Main()*). There may be times when you want to display all of this information to the screen. However, if your program is intended for people who are not very comfortable with computers, this may be a little confusing to them. If this is the case, you can use the instance variable *Message* instead. That is, just replace the Console.Out.WriteLine(e.ToString()); with Console.Out.WriteLine(e.Message);. Doing so will only display the message *Attempted to divide by zero* to the screen.

As you have seen, there are many different types of exceptions. Actually, the Exception class is what is called a *superclass*. That means that NumberFormatException and DivideByZeroException are two different kinds of exceptions. This is done through a process called *inheritance*. This course will not cover the topic of inheritance other than to say that every subclass has all of the characteristics of its superclass plus some that are special to the subclass.

I am bringing this up because I want you to know that in the above code, you could have replaced the line catch (Exception e) with catch (DivideByZeroException e), and the program would have functioned the same. The reason this is important is because you can actually have two different types of exceptions generated in the same try-block. Let me show you an example:

using System;

public class HandleGetInt

{

public static void Main()

{

String strNum;

int number;

int [] arrayA = new int[10];

Console.Out.Write("Enter a whole number: ");

strNum = Console.ReadLine();

try

{

number = Convert.ToInt32(strNum);

arrayA[number] = 100;

}

catch (FormatException e)

{

Console.Out.WriteLine("First catch-block");

Console.Out.WriteLine(e.Message);

}

catch (IndexOutOfRangeException e)

{

Console.Out.WriteLine("Second catch-block");

Console.Out.WriteLine(e.Message);

}

Console.Out.WriteLine("Goodbye!");

}

}

In this example, you will see that the program prompts the user for a whole number that will be used as an index into the arrayA variable. Notice there are two possible problems that can occur in this code. First, it is possible that the user will enter something other than a whole number. If this happens, a FormatException is thrown, and the program will leave the try-block and search for a catch-block for the FormatException. Once it finds this catch-block, it runs the code inside that block and then continues the program at the first line after all of the catch-blocks.

However, it is also possible that the user will enter a whole number, but this number may be out of range as an index of the array. If this occurs, then an IndexOutOfBoundsException is thrown. The program will exit the try-block and look for a catch-block that catches such an exception. Type this program in, and try typing in a variety of different things and see what the program does.

All of this exception handling might be interesting to you, but you also may be wondering how this can make your programs more useful. After all, in the previous examples, if the user enters the type of value or an invalid value, then your program just stops anyway. More than likely, you will want your program to continue to prompt the user for a value until it is valid. This can be done using a loop. Take the following program as an example:

using System;

public class ExceptionLoop

{

public static void Main()

{

String strNum;

int number;

bool isValid = false;

while (!isValid)

{

Console.Out.Write("Enter a whole number: ");

strNum = Console.ReadLine();

try

{

number = Convert.ToInt32(strNum);

isValid = true;

}

catch (FormatException e)

{

Console.Out.WriteLine(strNum + " is not a ¬

whole number.");

Console.Out.WriteLine("\n\nTry again.");

}

}

Console.Out.WriteLine("\n\nYou did it!");

}

}

Notice how this code will continue to prompt the user for a whole number until one is entered. Compile and execute this code to see how user-friendly it is. Now there is no way that the user can proceed with the program without typing in a whole number. Even better, the program does not crash if the user tries to enter invalid data.  
  
  
  
  
**Chapter 4**

**Input and Output Files**

Now I want to switch gears a little bit and take a look at files. To this point, all of your programs have taken input from the keyboard and directed output to the screen. While this is useful for most cases, if your program processes a lot of data, you will want to use data files. Let us first start with writing output data to a file.

To use a data file for output in C#, we must first create a StreamWriter object. We will then create a file and put the reference to that file into our newly created object. This is done by typing the following:

StreamWriter outFile = File.CreateText ¬

("output1.txt");

It is important to note that this line of code will create a new file named *output1.txt* in the same directory that the executable program resides. Also, you should know that if this file already exists, then the contents of that file will be overwritten. That means you should be careful when running your programs, because you may overwrite data that you want to keep. We can get around this, however. First, let's see an example of using files for output.

using System;

using System.IO;

public class Files

{

public static void Main()

{

StreamWriter outFile = File.CreateText

("output1.txt");

outFile.WriteLine("Hello file!");

outFile.Close();

}

}

I want to point out a few things to you about this program. First, notice that in addition to using the System namespace, this program is also using the System.IO namespace. The reason for this is because the StreamWriter class is defined in this namespace. Next, I want you to notice the line of code that creates our object and creates the new file. Also, you should note how we actually wrote our data to the file. We used the line, outFile.WriteLine("Hello file!");. This should look very familiar to you. In fact, the StreamWriter class has methods named *Write()* and *WriteLine()* that function exactly the same as the similar methods you have been using to display text on the screen. The final point I want to make about this code is the last line, which closes the file. It is good programming practice to always close a file when you are finished with it.

Type the above program in, and then compile and run your program. What happened? Usually when teaching students about output files, I have my students type in a very simple program such as this one and have them run it. Usually I will get a few people who run the program and tell me that the program didn't work. I ask them why they think that, and they usually respond, "Nothing printed on the screen." If this happened to you, I want you to open Notepad or another text editor and look for the output1.txt file. Open this file and see what is there. That's right. This file contains the text, "Hello file!" But why didn't anything display on the screen? The reason is because our code did not instruct the computer to do so. If we wanted the same text displayed on the screen, we should have typed, Console.Out.WriteLine("Hello file!");. Makes sense, doesn't it?

At this point, you may be wondering how we can get around destroying our file every time. After all, I'm sure you can imagine a scenario when you might want to keep the original contents of your file and simply add text to the end of the file. This is called *appending* to your file, and it can be done with the following line of code:

StreamWriter appFile = File.AppendText ¬

("output1.txt");

I want to remind you here that there is nothing special about my StreamWriter object names, outFile and appFile. I have just selected these names because they are descriptive. Really, the only thing that is different about this line of code is the fact that we are using the AppendText method instead of the CreateText method. Let's try this out with the following program:

using System;

using System.IO;

public class AppFiles

{

public static void Main()

{

StreamWriter appFile = File.AppendText

("output1.txt");

appFile.WriteLine("One more line of text.");

appFile.Close();

}

}

Again you will find that no output is displayed on the screen. However, when you open the output1.txt file, you should see a second line of text after the line that you saw before. I want to point out here that the AppendText method will open an existing file to which you can append text. In addition, if this file does not exist, the AppendText method will create the file just as the CreateText method did.

Now you should see how powerful it is to be able to write your output to a file. And of course, if there is a way to write to a file, there must be a way to read from a file. To do this, you need to create a StreamReader object and call on the OpenText method. Since there is only one way to read a file, as opposed to opening a blank input file or appending to an existing file, creating our object is a little easier. To do this, you can type:

StreamReader inFile = File.OpenText ¬

("output1.txt");

This code will create for us a StreamReader object named *inFile* and open the file output1.txt so that we can read data from the file into our program. The following program will use this code and open the file that you created in the previous example:

using System;

using System.IO;

public class OpenFiles

{

public static void Main()

{

String s;

StreamReader inFile = File.OpenText ¬

("output1.txt");

s = inFile.ReadLine();

Console.Out.WriteLine(s);

inFile.Close();

}

}

Again, notice how we were using the System.IO namespace and how there was a method named *ReadLine()* that is similar to the ReadLine() method we have been using. Type the program in, compile it, and run it; see what happens. This time, the file is opened, read from, and the text is displayed to the screen.

As you can see, reading from a file is very similar to writing to files. However, what do you think will happen if we attempt to read from a file that does not exist? If you attempt to do this, a FileNotFoundException will be generated, and the program will crash. Now with the knowledge from the previous chapters, you have the knowledge to catch this exception and keep your program from crashing. And while this is a good way to practice your exception skills, there is another way. C# provides a method called *Exists()* that returns true or false based on whether a file exists. So, now you can just use an *if* statement that calls on this method and does one thing if the file exists and something different if the file does not exist.

After introducing reading from and writing to files to my students, I always love to show them an interesting application of these principles: copying files. That's right. If you can write code that will open a file and read from it and open another file and write to it, then you can write a simple little program that will copy a text file. Let's look at the code:

using System;

using System.IO;

public class CopyFiles

{

public static void Main()

{

String s;

if (File.Exists("output1.txt"))

{

StreamReader inFile = File.OpenText ¬

("output1.txt");

StreamWriter outFile = File.CreateText ¬

("output2.txt");

s = inFile.ReadToEnd();

outFile.Write(s);

Console.Out.WriteLine("File copied!");

inFile.Close();

outFile.Close();

}

else

{

Console.Out.WriteLine("Input file does not ¬

exist.");

}

}

}

I want to point out in this example the use of the Exists() method to test to see if the input file exists. If it does, then the program will copy the data from the output1.txt file to the output2.txt file. This is done using the *ReadToEnd()* method instead of the ReadLine() method. As you might guess, this new method will read all of the text until it reaches the end of the stream. Try this program by compiling and running the code above. When you are finished running the program, look inside the directory with output1.txt, and you should have a new file named *output2.txt* that contains the same data.

A final point that I would like to make about the programs is regarding the names of our files. Have you noticed that all of these examples have had the actual name of file typed into the program? This is called *hard-coding* the program. This means that every time the program is run, the same files are used. If you wanted the program to open a different file, then the program would have to be rewritten and recompiled. This doesn't seem to be very flexible. However, notice that every time you have opened a file, the name of the file has been enclosed in double quotes. That means that a string is being passed to the method. Well if a literal string can be passed, then a string variable can be passed. Let's rewrite the last program that copies files, and this time, we will prompt the user for the new file name.

using System;

using System.IO;

public class CopyFiles2

{

public static void Main()

{

if (File.Exists("output1.txt"))

{

StreamReader inFile = File.OpenText ¬

("output1.txt");

Console.Out.Write("What name do you ¬

want for the new file? ");

String fileName = Console.ReadLine();

StreamWriter outFile = File.CreateText ¬

(fileName);

String s = inFile.ReadToEnd();

outFile.Write(s);

Console.Out.WriteLine("File copied!");

inFile.Close();

outFile.Close();

}

else

{

Console.Out.WriteLine("Input file does not ¬

exist.");

}

}

}

This code allows the user to give the new file any name they wish. Try this code for yourself. And of course, if you can use a variable to store the name of an output file, then you can certainly use a variable to store the name of an input file.  
  
  
  
  
**Chapter 5**

**Summary**

This lesson has given you some very powerful tools to use as an object-oriented programmer. First you learned how to make your programs more reliable by handling exceptions that may occur while your program is running. Now instead of your programs crashing when the user tries to do something that they shouldn't, they can handle any problems that occur and keep running. This is useful when getting input that should be in a certain format, or accessing an element in an array, or even when doing simple math operations.

In addition to learning about exceptions, you also learned how to make use of input and output files. You learned how to create new files for output as well as how to add new output to files that already exist. It is natural to immediately think about input files along with output files, and this lesson showed you how to do this.

The final lesson of this course will focus on graphical user interface (GUI) objects. You will learn how to create some objects that are commonly seen in an application that uses a GUI. This will make your programs more attractive, easier to use, more fun to work with, and more impressive to others.  
  
  
  
**Supplementary Material**

|  |
| --- |
| [TextReader and TextWriter](http://www.c-sharpcorner.com/UploadFile/ggaganesh/TextReaderNWriterGAG12052005235609PM/TextReaderNWriterGAG.aspx)  http://www.c-sharpcorner.com/UploadFile/ggaganesh/TextReaderNWriterGAG12052005235609PM/TextReaderNWriterGAG.aspx |
| The page will show you a different way to do input and output with files. |

**FAQs**   
  
**Q:** Sometimes we catch Exceptions and other times we catch FormatExceptions. What is the difference? Does it matter which we catch?  
  
**A:** The answer is that the Exception class is a very general type of Exception. Actually FormatException is a specific type of Exception. If you only have one type of Exception, it is ok to catch an Exception object. However, if you have a block of code that could throw more than one type of Exception, you will probably want special code to run for each type of Exception. For example if you have a block of code that could throw a FormatException would probably want to reprompt the user for a value. But if that same block of code throws a FileNotFoundException, then you probably want to prompt the user for a new file name. The only way to do that is with two different catch blocks, one for each of the specific Exception type.

**Assignment**   
  
  
Write a program that will keep track of movies you have seen using files. In addition to storing the name of the movie (a string), also keep track of the movie's rating (int) as a number of stars between 1 and 5. To make your program more flexible, use a file to keep track of the data so that it can be added to in the future. Here is a suggestion on how to make your program work:

When Main starts, it should open the data file and read in the movie data into two arrays, one for the name and one for the rating. Next have your program display a menu with choices to

1. display movie data
2. add movie data
3. quit

When the user chooses to quit, copy the data from the arrays to the data file. To make your program more flexible, be sure to handle any exceptions that may be thrown from invalid input.

If you want to practice more of your C# skills, write a class called Movie with two instance variables. Change your Main program so that it keeps track of the data using a Movie array variable. You could also add an option to the menu to delete a movie. You now have the knowledge to make very useful programs. Have fun with this!

[Click here for solution: **MovieFileData.zip**](https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/MovieFileData.zip)

|  |
| --- |
| Copyright © 1997 - 2012 Education To Go. All rights reserved. The material on this site cannot be reproduced or redistributed unless you have obtained prior written permission from Education To Go. Education To Go and ed2go are registered trademarks of Education To Go, a part of Cengage Learning. |

Bottom of Form