**try/catch Blocks**

When exceptions are thrown, you need to be able to handle them. This is done by implementing a *try/catch* block. Code that could throw an exception is put in the *try* block and exception handling code goes in the *catch* block. Listing 15-1 shows how to implement a *try/catch* block. Since an *OpenRead()*method could throw one of several exceptions, it is placed in the *try* block. If an exception is thrown, it will be caught in the *catch* block. The code in Listing 15-1 will print message and stack trace information out to the console if an exception is raised.

**Note:** The programs in this lesson cause exceptions on purpose. The exception that you see is generated intentionally to show you what the exception message looks like before you see it yourself in your own programs.

**Listing 15-1. Using try/catch Blocks: tryCatchDemo.cs**

using System;  
using System.IO;  
  
class tryCatchDemo  
{  
    static void Main(string[] args)  
    {  
        try  
        {  
            File.OpenRead("NonExistentFile");  
        }  
        catch(Exception ex)  
        {  
            Console.WriteLine(ex.ToString());  
        }  
    }  
}

Although the code in Listing 15-1 only has a single *catch* block, all exceptions will be caught there because the type is of the base exception type "Exception". In exception handling, more specific exceptions will be caught before their more general parent exceptions. For example, the following snippet shows how to place multiple catch blocks:

        catch(FileNotFoundException fnfex)  
        {  
            Console.WriteLine(fnfex.ToString());  
        }  
        catch(Exception ex)  
        {  
            Console.WriteLine(ex.ToString());  
        }

If the file doesn't exist, a *FileNotFoundException* exception will be thrown and caught by the first *catch* block. However, if a *PathTooLongException*exception was raised, the second catch part would catch the exception. This is because there isn't a *catch* block for the *PathTooLongException*exception and the generic *Exception* type *catch* block is the only option available to catch the exception.

Exceptions that are not handled will normally bubble up the stack until a calling routine in the call chain handles them. If you forget to include *try/catch*blocks in a part of your code and there aren't any *try/catch* blocks earlier in the call chain, your program will abort with a message describing the exception. To your users this would be very cryptic and uncomfortable. It is good practice to provide exception handling in your programs.

**Finally Blocks**

An exception can leave your program in an inconsistent state by not releasing resources or doing some other type of cleanup. A *catch* block is a good place to figure out what may have gone wrong and try to recover, however it can't account for all scenarios. Sometimes you need to perform clean up actions whether or not your program succeeds. These situations are good candidates for using a *finally* block.

Listing 15-2 illustrates the usefulness of a *finally* block. As you know, a file stream must be closed when you're done with it. In this case, the file stream is the resource that needs to be cleaned up. In Listing 15-2, *outStream* is opened successfully, meaning the program now has a handle to an open file resource. When trying to open the *inStream*, a *FileNotFoundException* exception is raised, causing control to go immediately to the *catch* block.

It's possible to close the *outStream* in the catch block, but what if the algorithm executed successfully without an exception? On success, the file would never be closed. Fortunately, we've included a *finally* block in Listing 15-2, which will always be executed. That's right, regardless of whether the algorithm in the *try* block raises an exception or not, the code in the *finally* block will be executed before control leaves the method.

**Listing 15-2. Implementing a finally Block: FinallyDemo.cs**

using System;  
using System.IO;  
  
class FinallyDemo  
{  
    static void Main(string[] args)  
    {  
        FileStream outStream = null;  
        FileStream inStream = null;  
  
        try  
        {  
            outStream = File.OpenWrite("DestinationFile.txt");  
            inStream = File.OpenRead("BogusInputFile.txt");  
        }  
        catch(Exception ex)  
        {  
            Console.WriteLine(ex.ToString());  
        }  
        finally  
        {  
            if (outStream != null)  
            {  
                outStream.Close();  
                Console.WriteLine("outStream closed.");  
            }  
            if (inStream != null)  
            {  
                inStream.Close();  
                Console.WriteLine("inStream closed.");  
            }  
        }

class multicatch

{

public static void main(String[] args)

{

int[] c={1};

String s="this is a false integer";

try

{

int x=5/args.length;

c[10]=12;

int y=Integer.parseInt(s);

}

catch(ArithmeticException ae)

{

System.out.println("Cannot divide a number by zero.");

}

catch(ArrayIndexOutOfBoundsException abe)

{

System.out.println("This array index is not accessible.");

}

catch(NumberFormatException nfe)

{

System.out.println("Cannot parse a non-integer string.");

}

}

}

|  |  |
| --- | --- |
| **Exception Class** | **Description** |
| System.IO.IOException | Handles I/O errors. |
| System.IndexOutOfRangeException | Handles errors generated when a method refers to an array index out of range. |
| System.ArrayTypeMismatchException | Handles errors generated when type is mismatched with the array type. |
| System.NullReferenceException | Handles errors generated from deferencing a null object. |
| System.DivideByZeroException | Handles errors generated from dividing a dividend with zero. |
| System.InvalidCastException | Handles errors generated during typecasting. |
| System.OutOfMemoryException | Handles errors generated from insufficient free memory. |
| System.StackOverflowException | Handles errors generated from stack overflow. |

try

{

// Code to try here.

}

catch (System.Exception ex)

{

// Code to handle exception here.

}

C#

try

{

// Code to try here.

}

finally

{

// Code to execute after try here.

}

C#

try

{

// Code to try here.

}

catch (System.Exception ex)

{

// Code to handle exception here.

}

finally

{

// Code to execute after try (and possibly catch) here.

}

Type of Exeption

[**ArgumentException**](http://www.dotnetperls.com/argumentexception)

[**ArgumentNullException**](http://www.dotnetperls.com/argumentnullexception)

[**ArgumentOutOfRangeException**](http://www.dotnetperls.com/argumentoutofrangeexception)

[**ArrayTypeMismatchException**](http://www.dotnetperls.com/arraytypemismatchexception)

[**DirectoryNotFoundException**](http://www.dotnetperls.com/directorynotfoundexception)

[**DivideByZeroException**](http://www.dotnetperls.com/dividebyzeroexception)

[**FileNotFoundException**](http://www.dotnetperls.com/filenotfoundexception)

[**FormatException**](http://www.dotnetperls.com/formatexception)

[**IndexOutOfRangeException**](http://www.dotnetperls.com/indexoutofrangeexception)

[**InvalidCastException**](http://www.dotnetperls.com/invalidcastexception)

[**InvalidOperationException**](http://www.dotnetperls.com/invalidoperationexception)

[**IOException**](http://www.dotnetperls.com/ioexception)[**KeyNotFoundException**](http://www.dotnetperls.com/keynotfoundexception)

[**NotImplementedException**](http://www.dotnetperls.com/notimplementedexception)

[**NullReferenceException**](http://www.dotnetperls.com/nullreferenceexception)

[**OutOfMemoryException**](http://www.dotnetperls.com/outofmemoryexception)

[**OverflowException**](http://www.dotnetperls.com/overflowexception)

[**StackOverflowException**](http://www.dotnetperls.com/stackoverflowexception)

[**TypeInitializationException**](http://www.dotnetperls.com/typeinitializationexception)