Exceptions

Objectives

- Use exceptions to handle unexpected but predictable undesirable situations while your program runs.
- Create exception handlers.
- Manage the unwinding of the stack in the presence of exceptions.
- Set the Message and Help properties of exceptions.
- Create custom exceptions.
- Manage nested exceptions.

Throwing and Catching Exceptions

C# developers distinguish among three things that might go wrong while their code is running:

- Bugs
- User errors
- Exceptions

A bug is a mistake in your code. Bugs should be found through careful testing, and repaired as quickly as possible. While few modern complex programs can claim to be bug-free, the only strategy for dealing with bugs is to squish them as quickly as possible.

User errors must be anticipated and coped with. When you ask the user to enter his name, he may well enter his birthdate. When you ask the user to click a button, he may right-click, click the wrong button, or pull the plug out of his computer. Users are like that, and you have to be ready and you must recover from their errors as well as you can.

Exceptions, however, are neither bugs nor a result of user errors. Exceptions are situations that are predictable, undesirable, and unpreventable. For example, when you try to read a file from the operating system, the file may not exist. Or the file may be in use by another application. Or you may not have enough memory to open the file. Each of these is an exceptional, but predictable circumstance.

In C#, exceptions encapsulate this concept of an exceptional circumstance, and they offer you an object-oriented way to handle the exception and to recover gracefully.

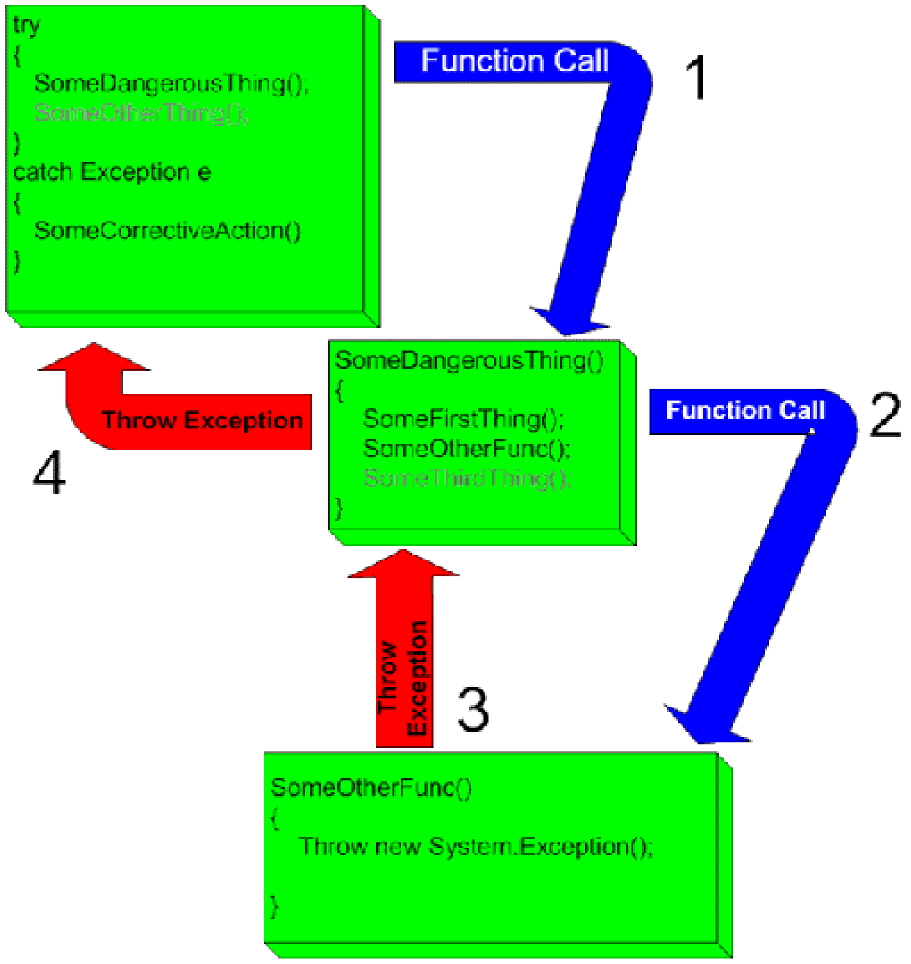
Exception Terminology

When an exceptional circumstance arises, an exception object is *thrown* (some programmers talk about *raising* an exception, but the common C# term is to *throw* the exception). When an exception is thrown, an exception handler *catches* the exception and takes corrective action.

When the exception is thrown, all execution stops. If there is an exception handler in the same method, then processing switches to the exception handler. If not, then the stack is *unwound* until a handler is found.

When the stack is unwound, control passes to the method that called the method that threw the exception. If that calling method has no handler, the stack is unwound further, to the method that called *that* method, and so forth.

This process of unwinding is illustrated in Figure 1. The illustration starts in the upper left corner. A method *SomeDangerousThing* is called (see 1). Within SomeDangerousThing, a second method, *SomeOtherFunc* is invoked (see 2). Within SomeOtherFunc an exception is thrown. There is no handler in SomeOtherFunc, so the stack is unwound and the exception is passed to the calling method, SomeDangerousThing (see 3). SomeDangerousThing does not have a handler, so the stack is further unwound, and the exception is passed to the first method (see 4). This first method *does* have a handler (see the catch block). The exception is handled, and program flow continues in this outermost method just below the catch block.



```
try
{
  SomeDangerousThing();
  SomeOtherThing();
}
catch Exception e
{
  SomeCorrectiveAction()
}
```

```
SomeDangerousThing()
{
  SomeFirstThing();
  SomeOtherFunc();
  SomeThirdThing();
}
```

```
SomeOtherFunc()
{
  Throw new System.Exception();
}
```

Figure 1. Unwinding the stack.

System.Exception

When you catch an exception, you may catch it by type. This allows you to narrow your exception handler down to a specific *type* of exception. For example, you might create an exception handler to catch all arithmetic exceptions with this code:

```
catch(System.ArithmeticException)
{ // … }
```

System.ArithemeticException derives from System.Exception and is supplied by the .NET Framework. Put the type of the exception in parentheses after the catch statement, and put the handler inside braces, as shown in the previous code.

The .NET Framework provides System.Exception, and a number of derived types. You are free to derive your own types from System.Exception as well. In any case, all exceptions must be of type System.Exception or a class derived from System.Exception.

If you do not provide a type for the catch statement, it will catch exceptions of any type at all:

```
catch
{ // … }
```

The code above is identical in its effect to writing:

```
catch(System.Exception)
{ // … }
```

Either of these statements will catch any exception that comes their way.

Exception Handling Order

Because exception handlers are evaluated in the order they appear in the method, you will want to put your more specialized exception handlers before your more general exception handlers.

Since System.DivideByZeroException derives from System.ArithmeticException, you will want to place the

DivideByZeroException handler *before* the more general ArithmeticException handler:

```
catch (System.DivideByZeroException) { // … }
catch (System.ArithmeticException) { // … }
catch { // … }
```

If the order were reversed, the DivideByZeroException would never be called.

```
catch (System.ArithmeticException) { // … }
catch (System.DivideByZeroException) { // … }
catch { // … }
```

In the order shown here, if a DivideByZero exception were to be raised, the exception would match the more general ArithmeticException and would be handled in that catch block. The DivideByZero handler would never run.

If you put these back into the correct order,

```
catch (System.DivideByZeroException) { // … }
catch (System.ArithmeticException) { // … }
catch { // … }
```

the first handler catches a DivideByZeroException. If another arithmetic exception is thrown, it will be ignored by DivideByZero and handled by the second handler. All other exceptions will be ignored by the first and second handler, but caught by the third.

Closing Resources

C# has garbage collection, so you don't have to worry that objects once created might hang around in memory if an exception takes you out of the normal program flow. With that said, there are other resources that are not managed by the garbage collector and can be left in an unacceptable state if an exception is thrown.

Imagine, for example, that in a method you open a file, take action, and then close the file. It is imperative that you close the file so that other code can get to the file. If, however, that action throws an exception, there is a danger that the file will not be closed:

```
SomeMethod()
{
   OpenFile();
   TakeDangerousAction();
   CloseFile(); // will you get here??
}
```

If the method TakeDangerousAction throws an exception (or calls a method that throws an unhandled exception) the call to CloseFile may never take place and the file may be left open.

You could try to fix this by adding a catch block, and closing the file in the catch block, but this method is prone to error.

```
SomeMethod()
{
   try
   {
      OpenFile();
      TakeDangerousAction();
      CloseFile(); // will you get here??
   }
   catch
   {
      CloseFile(); // error prone
   }
}
```

The problem with this approach is that changes made in the try block must be duplicated in the catch block. You might add additional catch blocks (for specialized errors) and each will need the call to CloseFile. Sooner or later you'll forget to update one or the other and your program will stop working. Or worse, it will work for a long time and then fail unexpectedly.

What you need is a way to ensure that the CloseFile() method will be called whether or not an exception is thrown. That is what the *finally* block is for. The code in a finally block is guaranteed to be called whether or not the catch block is called.

```
SomeMethod()
{
   try
   {
      OpenFile();
      TakeDangerousAction();
   }
   catch
   {




      CloseFile();   // guaranteed to get here
   }
}
```

Rules for finally Blocks

Finally blocks can be created without a catch block, but you must have a try block.

```
SomeMethod()
{
   try
   {
      OpenFile();
      TakeDangerousAction();
   }
   finally
   {
      CloseFile();   // guaranteed to get here
   }
}
```

Finally must not exit with break, continue, return, or goto. These unconditional branching statements are covered elsewhere in this course.

Try It Out!

*See **Exceptions.sln***

To see how to use exceptions, you'll create a simple test application and add exception handling.

1. Open a new console application and name it Exceptions.
2. Create a simple Tester class with a Run() method.
3. Within Run() display where you are and call Function1(). When you return from Function1() display that you are ready to exit from Run().

```
public class Tester
{
    public void Run()
    {
        Console.WriteLine("In Run()");
        Function1();
        Console.WriteLine("Exiting Run...");
    }
```

4. Add Function1. Have it invoke Function2. Before and after invoking Function1, display your progress.

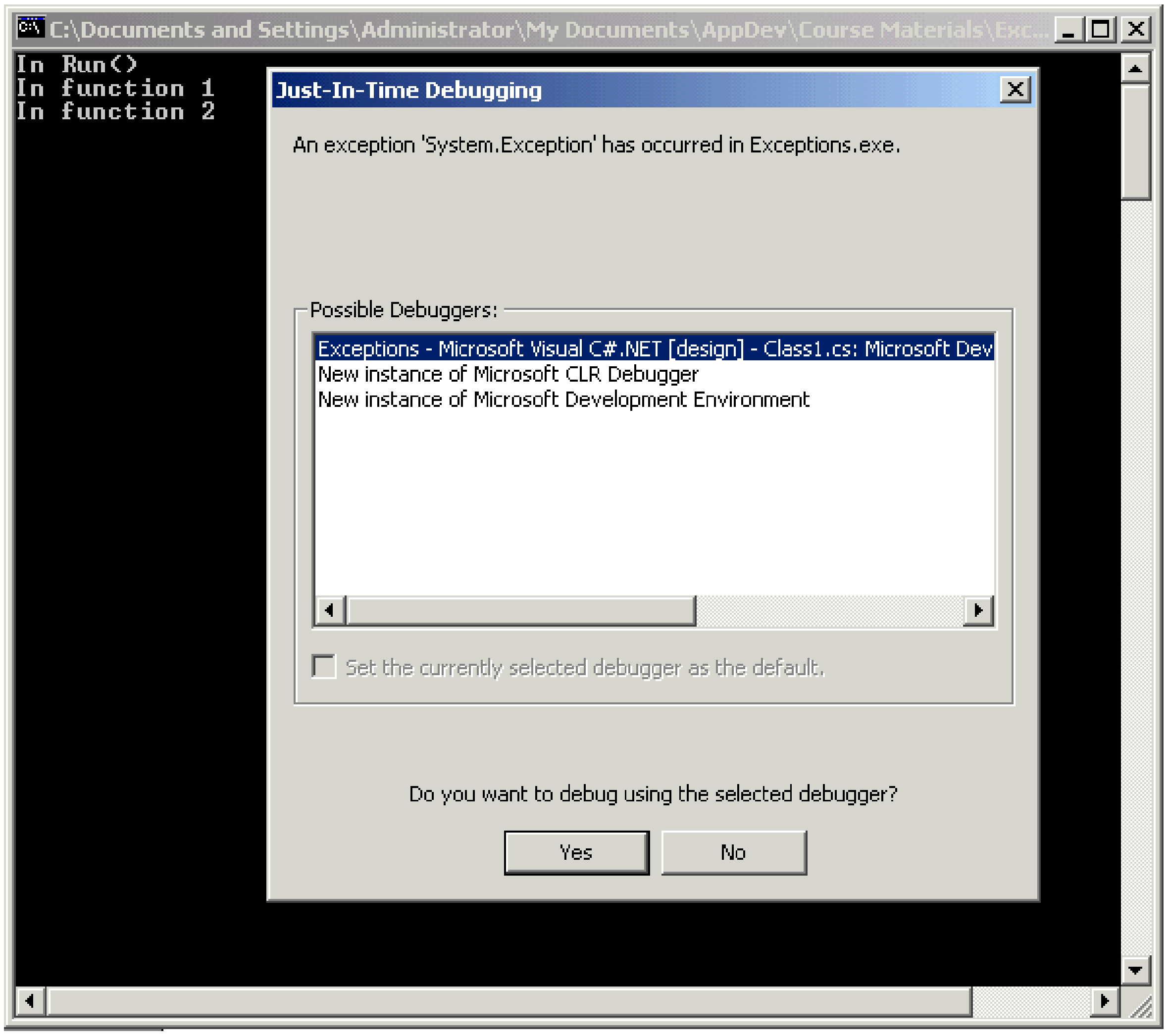
```
public void Function1()
{
    Console.WriteLine("In function 1");
    Function2();
            Console.WriteLine("Exiting function 1");
        }
```

5. Write Function2. Within Function2, display where you are, but throw an exception.

```
public void Function2()
{
   Console.WriteLine("In

   Console.WriteLine("Exiting function 2");
}
```

Notice that you throw an exception by creating one on the heap with the *new* keyword. In this case, you'll throw an instance of System.Exception, passing no parameters to the constructor.

6. Run the application. The results are shown in Figure 2. You can see that the Just In Time debugger has caught the exception. Looking at the output, you see the beginning of Run, Function1, and Function2, but none of the functions complete.


C:\Documents and Settings\Administrator\My Documents\AppDev\Course Materials\Exc...
Just-In-Time Debugging
Possible Debuggers:
Yes
No

```
In Run()
In function 1
In function 2
```

An exception 'System.Exception' has occurred in Exceptions.exe.

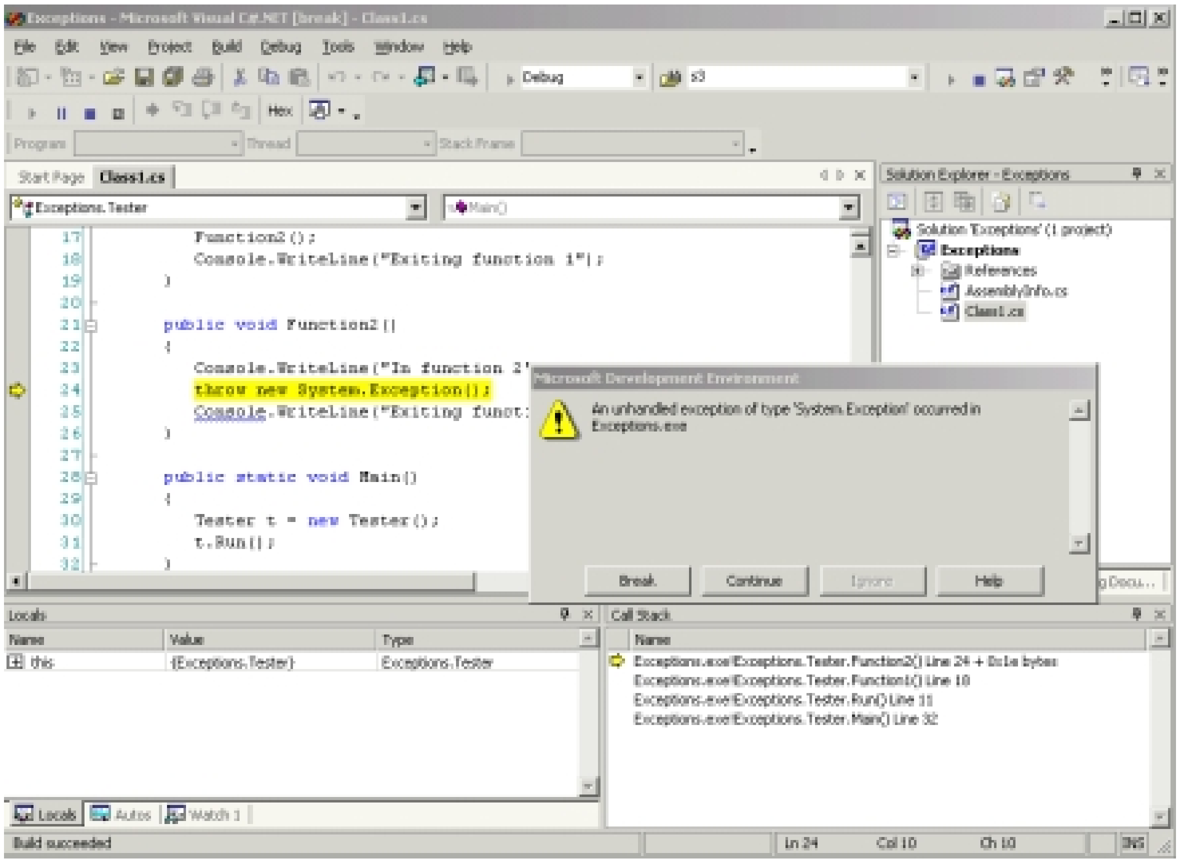
Exceptions - Microsoft Visual C#.NET [design] - Class1.cs: Microsoft Dev
New instance of Microsoft CLR Debugger
New instance of Microsoft Development Environment

☐ Set the currently selected debugger as the default.

Do you want to debug using the selected debugger?

Figure 2. Throwing the exception.

7. Click **Yes** and proceed into the debugger. You can get to the same place by running the code in the debugger (press **F5**). In either case, you find yourself in the development environment. The current line of execution is the line in which the exception is thrown. The Debugging dialog box offers you the option to break there or to continue, as shown in Figure 3. Notice in the lower right corner that the call stack shows that the static method Main called the instance method Run, which called Function1 which in turn called the current method: function2.



```
            Function2();
            Console.WriteLine("Exiting function 1");
        }

        public void Function2()
        {
            Console.WriteLine("In function 2'
            throw new System.Exception();
            Console.WriteLine("Exiting funct:
        }

        public static void Main()
        {
            Tester t = new Tester();
            t.Run();
        }
```

An unhandled exception of type 'System.Exception' occurred in Exceptions.exe

```
Exceptions.exe!Exceptions.Tester.Function2() Line 24 + 0x1a bytes
Exceptions.exe!Exceptions.Tester.Function1() Line 18
Exceptions.exe!Exceptions.Tester.Run() Line 11
Exceptions.exe!Exceptions.Tester.Main() Line 32
```

Figure 3. Catching the exception.

8. Click on **Break**, and press **SHIFT+F5** to stop debugging.

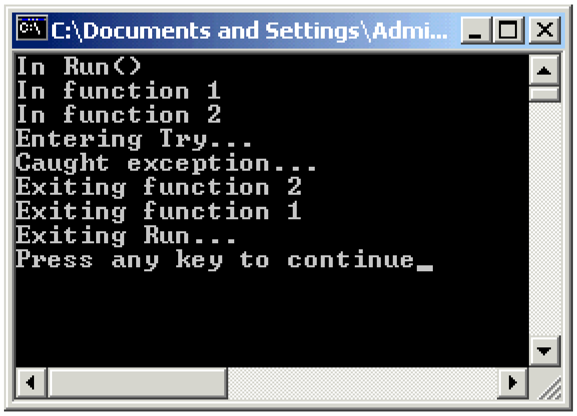
*See **Exceptions2.sln***

To avoid crashing, your program needs a catch block to handle the exception.

9. Rewrite Function2: surround the exception with a try block. Add a catch block to handle the exception; in this case you'll just display a message indicating that you caught the exception.

```
public void Function2(){
   Console.WriteLine("In function 2");

   try
   {
      Console.WriteLine("Entering Try...");
      throw new System.Exception();
      Console.WriteLine("Exiting Try...");
   }
   catch
   {
      Console.WriteLine("Caught exception...");
   }
   Console.WriteLine("Exiting function 2");
}
```

10. Run the application. The results are shown in Figure 4. The catch block allows your program to recover. You now see Function2 return, Function1 returns, and Run returns. All is right with the world.



```
In Run()
In function 1
In function 2
Entering Try...
Caught exception...
Exiting function 2
Exiting function 1
Exiting Run...
Press any key to continue_
```

Figure 4. Catching the exception.

Unwinding the Stack

Often you will not have a catch block (or even a try block) in the particular method that throws an exception. Unhandled exceptions are passed back to the calling method, and you may have an exception handler there.

Try It Out!

See ***Exceptions3.sln***

To see how a calling function might handle an exception, you'll rewrite the previous example.

1. Reopen the previous example.

2. Take the try/catch block out of function2.

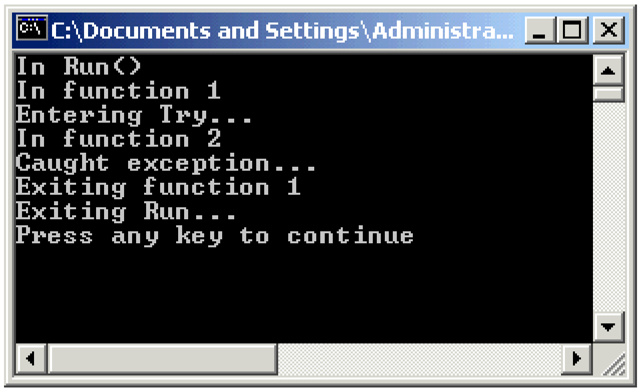
```
public void Function2()
{
   Console.WriteLine("In function 2");
   throw new System.Exception();
   Console.WriteLine("Exiting function 2");
}
```

3. Add a try/catch block to Function1. Function1 will now call Function2 from within its try block. If an unhandled exception is thrown in Function2, it will be caught in the try block of Function1.

```
public void Function1()
{
   Console.WriteLine("In function 1");
   try
   {
      Console.WriteLine("Entering Try...");
      Function2();
      Console.WriteLine("Exiting Try...");
   }
   catch
   {
      Console.WriteLine("Caught exception...");
   }
   Console.WriteLine("Exiting function 1");
}
```

4. Run the application. The results are shown in Figure 5. Examine Figure 5 carefully and compare it with Figure 4. They are different in a very important way. In Figure 4 you see that Function2 exits cleanly and the

message *Exiting Function2* is displayed. That message does not appear in Figure 5.



```
In Run()
In function 1
Entering Try...
In function 2
Caught exception...
Exiting function 1
Exiting Run...
Press any key to continue
```

Figure 5. Catching a calling function.

In this example, the method Function2 did not exit cleanly; an exception was thrown and not handled. The stack was unwound and the exception was handled in Function1, but the final line of Function2 never executes. This is a critical distinction.

Catching Specific Exception Types

Until now you've been using the generic System.Exception. The .NET Framework provides a number of specialized exception types to help you manage various exceptional circumstances.

Try It Out!

*See **Exceptions4.sln***

In the next example, you will create a method that under one circumstance throws a DivideByZero exception, and under a different circumstance throws an ArithmeticException. Your code must differentiate between these two exception types.

It is important to understand that DivideByZeroException derives from ArithmeticException, which in turn derives from System.Exception. Thus, DivideByZeroException is an ArithmeticException.

1. Reopen the previous example.

2. Create a new method, *DoDivide*. This method will take two doubles and return the quotient as a double. If the divisor is zero you will throw the specialized DivideByZeroException because it is illegal to divide by zero.

Normally, it is just fine to divide zero by another number, but in this case, you're not going to allow that, and you're going to throw the more general ArithmeticException. This is a bit contrived (since dividing zero by another number is mathematically just fine) but it will illustrate how exceptions are handled.

```
public double DoDivide (double dividend, double divisor)
{
   if (divisor == 0)
      throw new System.DivideByZeroException();
   if (dividend == 0)
      throw new System.ArithmeticException();
   return dividend/divisor;
}
```

3. You will call this method from Function2, The first time you call you'll divide 12 by 15.

```
public void Function2()
 {
    Console.WriteLine("In function 2");
    double x = 12;
    double y = 15;
    Console.WriteLine("Trying {0} / {1}...", x,y);
    Console.WriteLine ("{0} / {1} = {2}",
      x, y, DoDivide(x,y));
```

4. Assign the value zero to the local variable y and call DoDivide again.

```
Console.WriteLine("Trying {0} / {1}...", x,y);
Console.WriteLine ("{0} / {1} = {2}",
   x, y, DoDivide(x,y));
```

5. When you are done with Function2, display that and exit the method.

```
   Console.WriteLine("Exiting function 2");
}
```

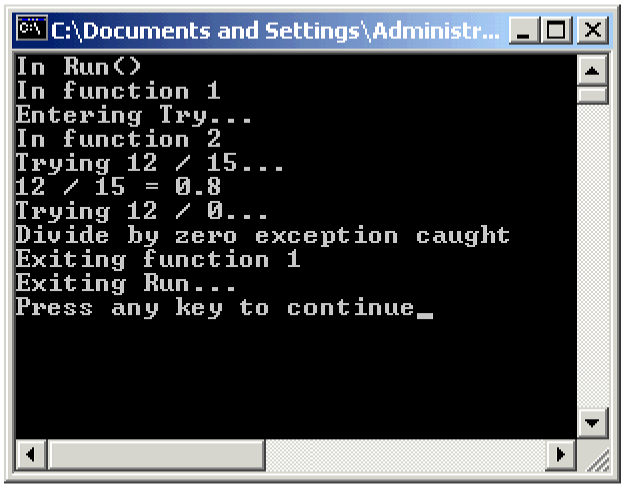
Notice that there is no try or catch block in Function2. You'll catch any exceptions in Function1.

6. Rewrite Function1 to call Function2 within a try block, but specialize your exception handling. You'll need three catch blocks. The first will catch a DivideByZeroException, the second will catch the more general ArithmeticException, and the third will catch any exception at all.

```
public void Function1()
{
   Console.WriteLine("In function 1");
   try
   {
      Console.WriteLine("Entering Try...");
      Function2();
      Console.WriteLine("Exiting Try...");
   }

   catch (System.DivideByZeroException)
   {
      Console.WriteLine(
         "Divide by zero exception caught");
   }
   catch (System.ArithmeticException)
   {
      Console.WriteLine(
         "ArithmeticException exception caught");
   }

   catch
   {
      Console.WriteLine(
         "Caught unknown exception...");
   }
   Console.WriteLine("Exiting function 1");
}
```

7. Run the program. The results are shown in Figure 6. The first time through, the DoDivide method works fine and no exception is thrown; *12/15 = 0.8*. The second time through, you try to divide 12 by 0 and that

does throw an exception. The exception is caught and the correct specific message is displayed.



C:\Documents and Settings\Administr...

```
In Run()
In function 1
Entering Try...
In function 2
Trying 12 / 15...
12 / 15 = 0.8
Trying 12 / 0...
Divide by zero exception caught
Exiting function 1
Exiting Run...
Press any key to continue_
```

Figure 6. Catching specialized exceptions.

8. Try reversing the catch blocks in Function1.

```
public void Function1()
{
   Console.WriteLine("In function 1");
   try
   {
      Console.WriteLine("Entering Try...");
      Function2();
      Console.WriteLine("Exiting Try...");
   }
   catch (System.ArithmeticException)
   {
      Console.WriteLine(
         "ArithmeticException exception caught");
   }
   catch (System.DivideByZeroException)
   {
      Console.WriteLine(
         "Divide by zero exception caught");
   }
```

```
    catch
    {
       Console.WriteLine("Caught unknown exception...");
    }
    Console.WriteLine("Exiting function 1");
}
```

9. Run the application again. The program won't compile. The error message, on the line for the DivideByZero exception is: *A previous catch clause already catches all exceptions of this or a super type (System.ArithmeticException).* The compiler recognizes that it is impossible to catch a DivideByZero exception if you place the ArithmeticExcpetion first!

Finally

In the previous example, Function1 calls Function2 and Funtion2 throws an exception. The catch block in Fuction1 catches the exception and does not crash, but the line after the call to Function2, *Exiting Try…* is never invoked. This line doesn't do anything important, so no real harm is done, but there are circumstances where not invoking the lines following the call to a function might be disastrous. For example, if Function1 were to open a file and then call Function2 and then close the file, the exception in Function2 might prevent the file from ever being closed.

You can solve that problem with a finally block. The keyword *finally* creates a block of code that will run *whether or not an exception is handled*. The finally block can only follow a try block: you can't have finally without try, though you can have finally without a catch block.

Try It Out!

See ***Exceptions5.sln***

In this next exercise, you'll rewrite the previous example to illustrate the use of the finally block.

1. Reopen the previous example.

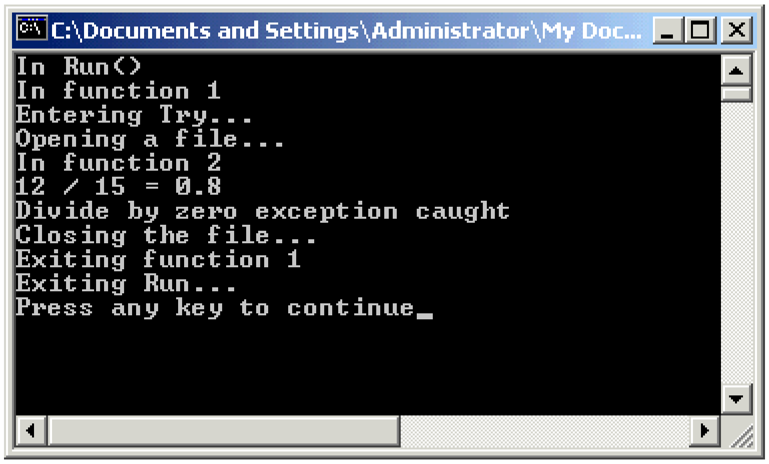
2. In the try block within Function1 open a file (you'll mimic this by displaying the words *Opening a file…*).

```
public void Function1()
{
   Console.WriteLine("In function 1");
   try
   {
      Console.WriteLine("Entering Try...");
      Console.WriteLine("Opening a file...");
      Function2();
      Console.WriteLine("Exiting Try...");
   }
catch (System.DivideByZeroException)
{
   Console.WriteLine("Divide by zero exception caught");
}
catch (System.ArithmeticException)
{
   Console.WriteLine("ArithmeticException exception
caught");
}

catch

   Console.WriteLine("Caught unknown exception...");
}
```

3. You must close the file. You can't put it in the try block because Function2 might throw an exception. Instead, you'll put it in the finally block where it is guaranteed to run *whether or not* there is an exception.

```
finally
{
   Console.WriteLine("Closing the file...");
}
```

5. Run the application. The results are shown in Figure 7. Notice that the finally block has executed even though an exception was caught.

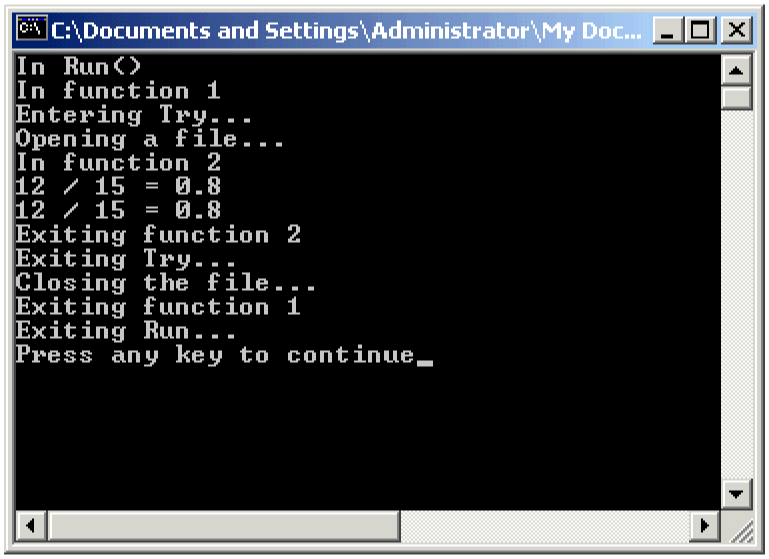


```
In Run()
In function 1
Entering Try...
Opening a file...
In function 2
12 / 15 = 0.8
Divide by zero exception caught
Closing the file...
Exiting function 1
Exiting Run...
Press any key to continue_
```

Figure 7. Executing the finally block.

6. Comment out the call to set y to 0. This removes the need to throw an exception.

7. Run the application again. The results are shown in Figure 8. This time no exception is thrown, and you see the display of *Exiting Try…*. Notice, however, that the message *Closing the file…* is still displayed; the *finally* block is invoked whether or not an exception is thrown.



```
In Run()
In function 1
Entering Try...
Opening a file...
In function 2
12 / 15 = 0.8
12 / 15 = 0.8
Exiting function 2
Exiting Try...
Closing the file...
Exiting function 1
Exiting Run...
Press any key to continue_
```

Figure 8. No exception was thrown.

The Exception Object

Until now you've been dealing with exceptions as sentinel objects; their very presence signals a problem. Exceptions are classes, however, and as such they have methods and properties. System.Exception, for example, has a few very useful properties.

The Message Property stores a message describing the exception. Each .NET Framework exception type has a default message, but you are free to set the message as well. One overloaded version of the constructor for System.Exception takes a *message* parameter; whatever string you pass in is stored as the message for that exception.

A second useful property is the HelpLink. This is a URL that you can store with the exception to guide the user to more documentation about the exception itself.

A third very helpful property is the StackTrace. By displaying the StackTrace the developer can examine the exact location in the code where the exception was thrown. You can take control of this information to write it to a log file or to display it to the user.

Try It Out!

See ***Exceptions6.sln***

To see how these properties are used, you'll rewrite the previous example to create and manipulate instances of .NET exceptions.

1. Reopen the previous example.

2. Rewrite DoDivide. If the divisor is 0, instantiate a *DivideByZeroException* object.

```
public double DoDivide (double dividend, double divisor)
{
   if (divisor == 0)
   {
      DivideByZeroException e =
         new DivideByZeroException();
```

3. Add a help link for this exception. Typically you'd provide a URL to a specific document. In this case, just provide a general URL to a Web site.

```
e.HelpLink = "http://www.LibertyAssociates.com";
```

4. Now that the exception has been set the way you want it, throw the exception.

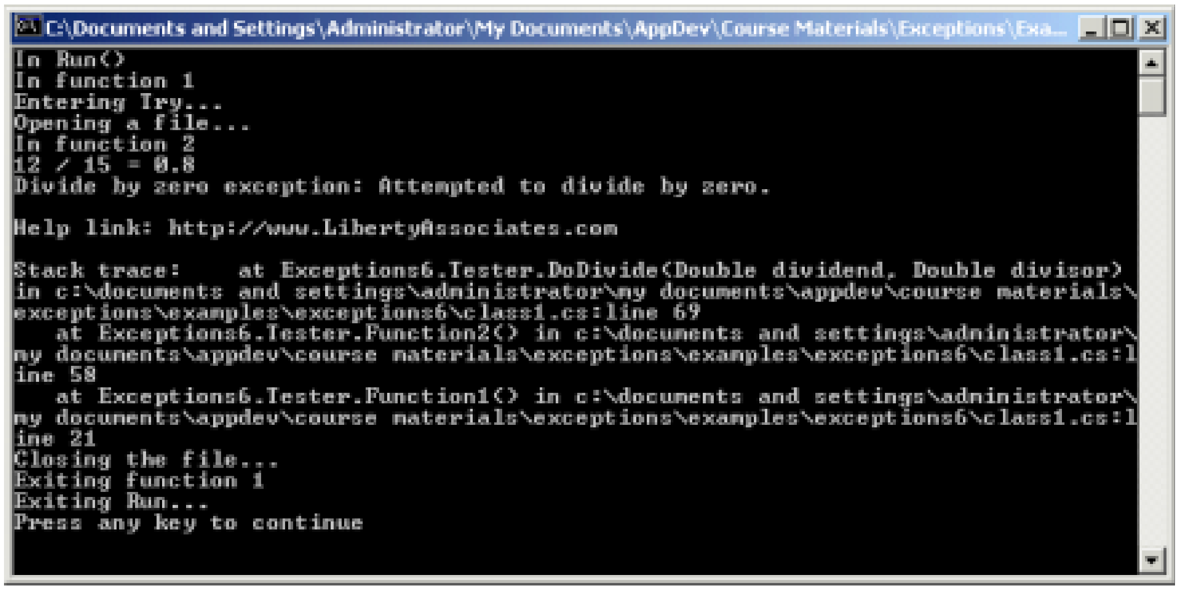
```
throw e;
```

5. Modify the code that catches the DivideByZero exception to display the default message, along with the HelpLink you created and the StackTrace created by the CLR.

```
catch (System.DivideByZeroException e)
{
   Console.WriteLine("Divide by zero exception: {0}",
      e.Message );

   Console.WriteLine("\nHelp link: {0}", e.HelpLink);

   Console.WriteLine("\nStack trace: {0}", e.StackTrace);
}
```

Notice that this time you provide an identifier for the DivideByZeroException. Here you use the letter *e* as the identifier; this is a common idiom. You can display the Message property by writing *e.Message*, just as you would with any other object.

6. Run the application; the results are shown in Figure 9. You can see the exception thrown, and the default message is displayed: *Attempted to divide by zero*. The HelpLink property is used to display a URL for more information, and the stack trace tells you the exact line on which the exception occurs. Notice that the stack trace also shows the line on which Function1 calls Function2, and on which Run calls Function1. This is a complete stack trace to help you track down exactly what was happening in your code.



```
C:\Documents and Settings\Administrator\My Documents\AppDev\Course Materials\Exceptions\Exa...
In Run()
In function 1
Entering Try...
Opening a file...
In function 2
12 / 15 = 0.8
Divide by zero exception: Attempted to divide by zero.

Help link: http://www.LibertyAssociates.com

Stack trace:    at Exceptions6.Tester.DoDivide(Double dividend, Double divisor)
in c:\documents and settings\administrator\my documents\appdev\course materials\
exceptions\examples\exceptions6\class1.cs:line 69
   at Exceptions6.Tester.Function2() in c:\documents and settings\administrator\
my documents\appdev\course materials\exceptions\examples\exceptions6\class1.cs:l
ine 58
   at Exceptions6.Tester.Function1() in c:\documents and settings\administrator\
my documents\appdev\course materials\exceptions\examples\exceptions6\class1.cs:l
ine 21
Closing the file...
Exiting function 1
Exiting Run...
Press any key to continue
```

Figure 9. Exception properties.

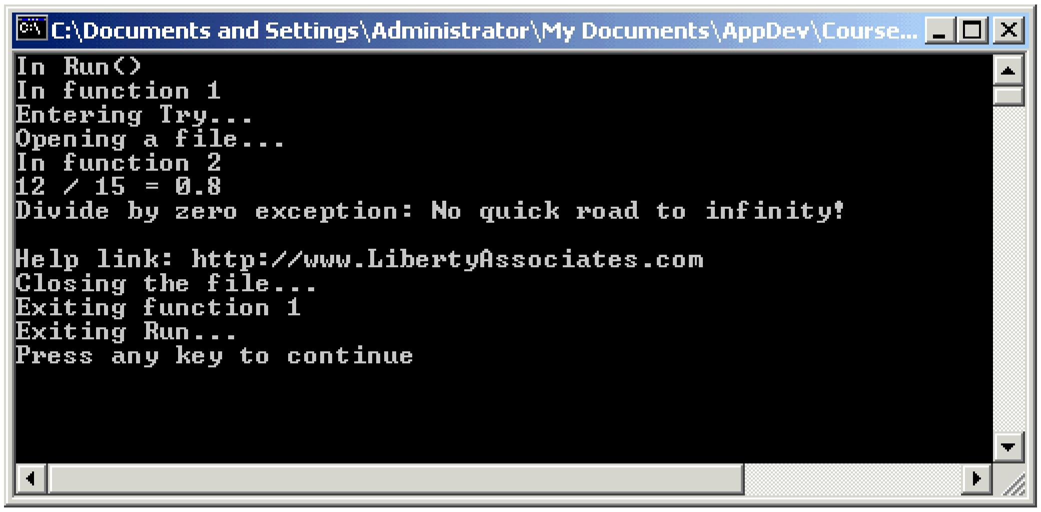
7. Modify the creation of the DivideByZero exception. This time, pass a custom message to the constructor.

```
public double DoDivide (double dividend, double divisor)
{
   if (divisor == 0)
   {
      DivideByZeroException e =
         new DivideByZeroException(
            "No quick road to infinity!");
```

8. Comment out the line with the stack trace (to simplify the output).

```
catch (System.DivideByZeroException e)
{
   Console.WriteLine("Divide by zero exception: {0}",
      e.Message );

   Console.WriteLine("\nHelp link: {0}", e.HelpLink);

  // Console.WriteLine("\nStack trace: {0}",
   // e.StackTrace);
}
```

9. Rerun the application. The results are shown in Figure 10. Your custom message is displayed. When you instantiate the DivideByZeroExceptionObject the string you pass as a parameter is sent up to the base class System.Exception and assigned to the Message property.


C:\Documents and Settings\Administrator\My Documents\AppDev\Course...

```
In Run()
In function 1
Entering Try...
Opening a file...
In function 2
12 / 15 = 0.8
Divide by zero exception: No quick road to infinity!

Help link: http://www.LibertyAssociates.com
Closing the file...
Exiting function 1
Exiting Run...
Press any key to continue
```

Figure 10. A custom exception message.

Creating Custom Exception Types

The .NET Framework provides a number of specialized exception types, but if you can't find what you want, you are free to create your own. You may then throw these exceptions and catch them just as you might throw and catch any of the built-in types.

The only restriction on your built-in types is that they must ultimately derive from System.Exception. You are free to derive from an existing exception type or from System.Exception directly.

Rethrowing Exceptions

So far, whenever you've thrown an exception, you've caught it in a catch block, handled it, and moved on. There are times, however, when you will take some action in the exception handler and then throw the exception again. You can rethrow the same exception just by writing.

```
throw;
```

You might decide to throw a new exception, however. You do that by creating the new exception and throwing it from within the exception handler for the first exception.

Inner Exceptions

Throwing a new exception is fine, but perhaps you don't want to lose the information stored in the original exception. You can wrap the old exception inside the new exception! Every exception has a reference to another exception, known as the inner exception. When your exception is created the InnerException property is null, but you are free to assign an exception to it.

For example, you might catch a DivideByZero exception and take some action. You might then create a new System.Exception object to throw, but you'll nest the original DivideByZero inside it to preserve the original message, as shown in Figure 11. The original DivideByZero exception is housed within the new System.Exception object.

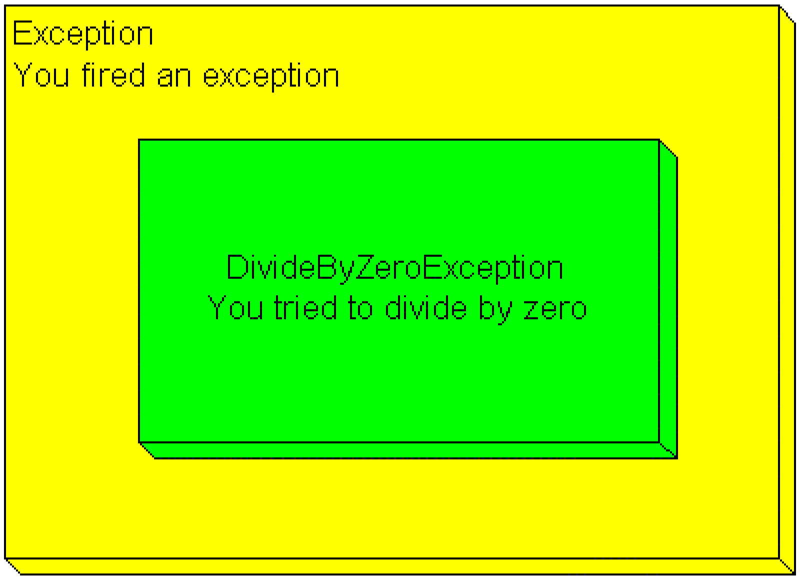


Figure 11. Nesting an exception.

When you catch the System.Exception object you might decide to throw a new exception, nesting the original exception within the new, custom exception. Of course, the Exception object you are nesting is itself nesting the DivideByZero object, as shown in Figure 12.

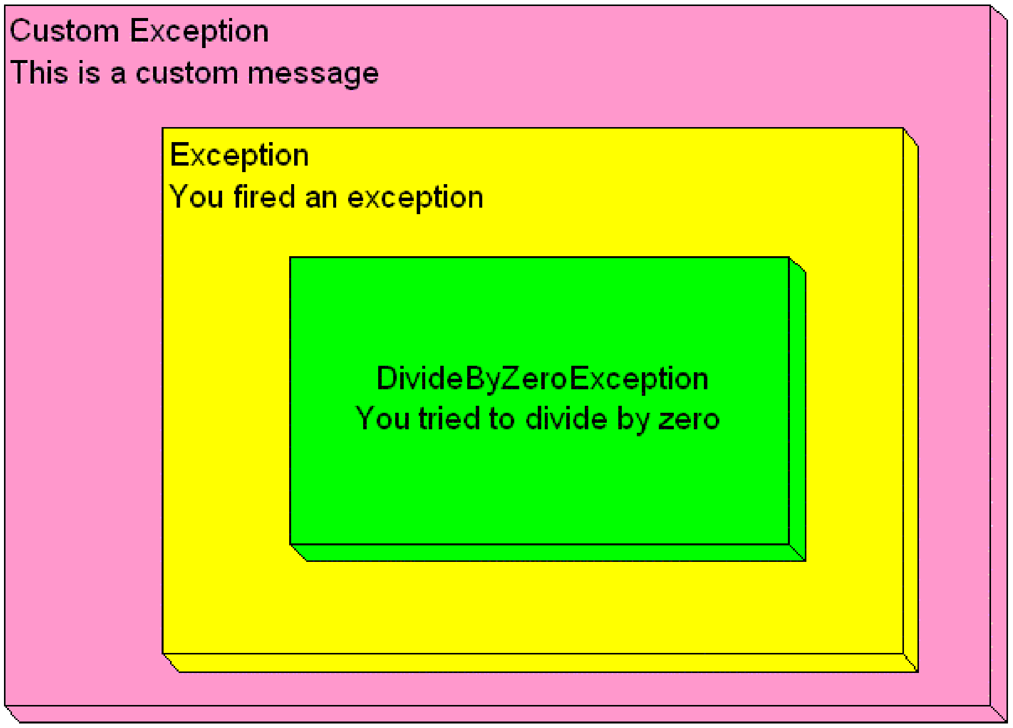


Figure 12. Cascading nested exceptions.

Try It Out!

*See **Custom Exceptions.sln***

To see how custom exceptions work, you'll create a simple console application and add a custom exception type. You'll also nest exceptions to see the advantage of keeping an exception history.

1. Create a new console application named CustomExceptions.
2. Create a custom exception type, *BozoException*. The constructor will take two parameters: a string *message* and an object of type System.Exception *inner*.

```
public class BozoException : System.Exception
{
   public BozoException(
      string message,
      Exception inner):
      base(message, inner)
   { }
}
```

BozoException derives from *System.Exception*, and it chains up to the Exception constructor, passing along the two parameters to its base class' constructor. System.Exception's constructor is overloaded to take these two objects. The first, *message* is assigned to the Message property. The second, *inner,* is assigned to the InnerException property. This is how inner exceptions are stored.

3. Create a Tester class with seven methods, each of which uses Console.WriteLine to display its progress. The first method is the static method Main, which instantiates a Tester and calls Run.

```
public static void Main()
{
   Console.WriteLine("In Main()");
   Tester t = new Tester();
   t.Run();
   Console.WriteLine("Exiting Main...");
}
```

4. Create the Run method. Other than display its progress, Run's only job is to invoke Method1.

```
public void Run()
{
   Console.WriteLine("In Run()");
   Method1();
   Console.WriteLine("Exiting Run...");
}
```

5. Method1 opens a file and then invokes Method 2.

```
Console.WriteLine("In Method 1");
try
{
   Console.WriteLine("Entering Try...");
   Console.WriteLine("Opening a file...");
   Method2();
   Console.WriteLine("Exiting Try...");
}
```

6. Method2 invokes Method3, which invokes Method4, which in turn invokes Method5. Method5 throws a DivideByZeroException, passing in a custom message.

```
public void Method5()
{
   throw new DivideByZeroException(
      "Method5 Divided by zero exception");
}
```

7. The stack is unwound and the exception is caught by Method4 (the method that called Method5). Give Method4 a catch block to catch ArithmeticException objects.

Method4 will take partial action based on this exception. In this case it just prints a message to the console, but in a real application it might take other actions based on registering the exception. It then rethrows the exception, exactly as is, making no modifications to it.

```
public void Method4()
{
   try
   {
      Method5();
   }
   catch (System.ArithmeticException e)
   {
      Console.WriteLine(
         "Method4 caught an arithmetic
         exception. Rethrowing...");
      throw; // just toss it back out there
   }
}
```

Notice that the catch block is looking for a base class of DivideByZeroException. Since DivideByZeroException derives from ArithmeticException it *is-an* ArithmeticException and so the catch block matches. Note also that in order to rethrow the exception you use the *throw* keyword.

8. The stack is unwound and the exception is offered to Method3. Method3 has a handler for DivideByZero exception. In the handler Method3 creates a new exception object (in this case, of type System.*Exception*). It creates the new exception object and passes in two parameters: a string and the original exception. The original exception is assigned to the *InnerException* property of the new exception, then the new exception is thrown.

```
public void Method3()
{
   try
   {
      Method4();
   }
   catch (System.DivideByZeroException e)
   {
      Exception ex =
         new Exception(
            "Method3 Caught divide by zero", e);
      throw ex;
   }
}
```

9. Once again the stack is unwound, this time to Method2. Method2 has a catch block for System.Exception, and so catches the thrown exception. In the exception handler, it creates a new instance of the custom exception type BozoException, passing in a new message and the old exception.

Notice that the exception it passes in is the System.Exception object created in Method3's catch block, which in turn has as a nested exception, the original DivideByZeroException object. This System.Exception object is now the InnerException property in the new BozoException.

```
public void Method2()
{
   try
   {
      Method3();
   }
   catch (System.Exception e)
   {
      BozoException bozo =
         new BozoException(
            "Method2 Custom Exception Situation!",e);
      throw bozo;
   }
}
```

10. The catch block then throws this BozoException, again unwinding the stack. Control returns to the catch block for Method1. In that catch block, you will print out the message from the exception and all its nested exceptions.

```
public void Method1()
{
   Console.WriteLine("In Method 1");
   try
   {
      Console.WriteLine("Entering Try...");
      Console.WriteLine("Opening a file...");
      Method2();
      Console.WriteLine("Exiting Try...");
   }

   catch (BozoException e)
   {
      Console.WriteLine("\n{0}",e.Message);
      Console.WriteLine("Exception history...");
      Exception inner = e.InnerException;
      while (inner != null)
      {
         Console.WriteLine("{0}", inner.Message);
         inner = inner.InnerException;
      }
   }
   finally
   {
      Console.WriteLine("Closing the file...");
   }
   Console.WriteLine("Exiting Method 1");
}
```

You start by printing the message from the exception:

```
catch (BozoException e)
{
   Console.WriteLine("\n{0}",e.Message);
```

To print the inner exception messages, you start by creating a local object of type Exception and assigning to it the InnerException property of the exception you caught.

```
  Exception inner = e.InnerException;
```

You then create a while loop to test if *inner* is null. You print the message, and then you assign to inner its own InnerException property, thus reassigning the nested exception object to inner.

```
while (inner != null)
{
   Console.WriteLine("{0}", inner.Message);
   inner = inner.InnerException;
}
```

This is a tricky bit of code, so make sure you understand how it works before going forward. At the start of this catch block you have the following structure:

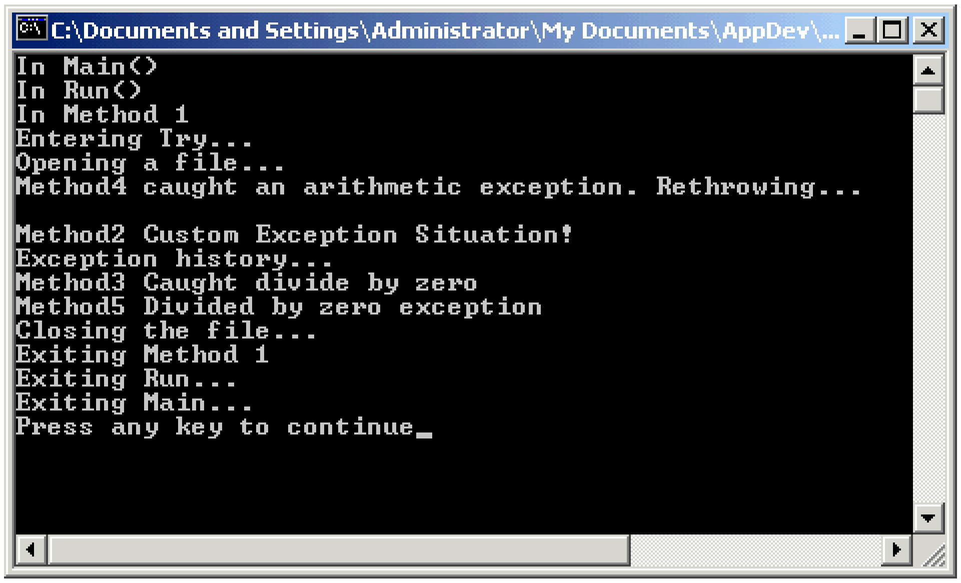
```
BozoException
   System.Exception
      DivideByZeroException
```

BozoException has an InnerException property of type System.Exception, which in turn has an InnerException property of type DivideByZeroException. When you assign *inner* to e.InnerException, e is the BozoException and *inner* is now a reference to System.Exception. When you write

```
   inner = inner.InnerException
```

inner is the System.Exception so InnerException is the DivideByZeroException. At the end of the assignment, *inner* refers to the DivideByZeroException!

11. Run the application. The results are shown in Figure 13. You see the exception history printed, much as you might expect.

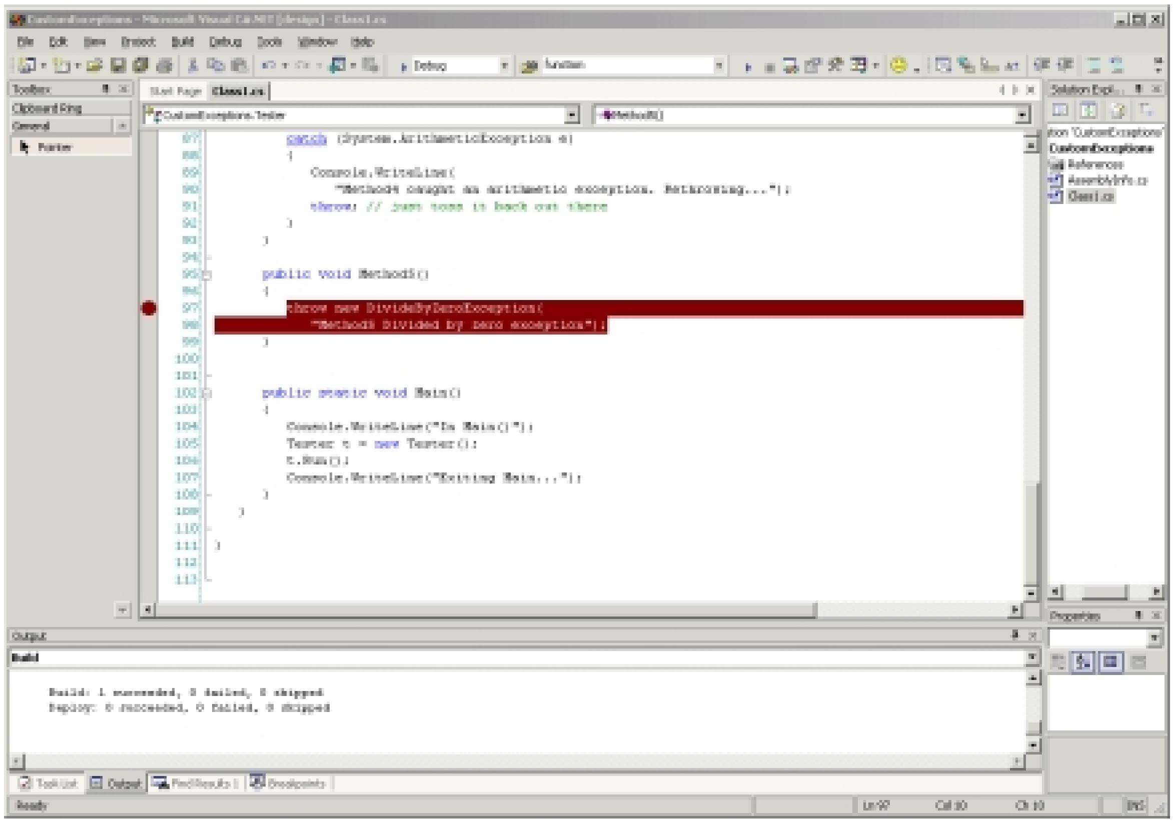


```
C:\Documents and Settings\Administrator\My Documents\AppDev\...
In Main()
In Run()
In Method 1
Entering Try...
Opening a file...
Method4 caught an arithmetic exception. Rethrowing...

Method2 Custom Exception Situation!
Exception history...
Method3 Caught divide by zero
Method5 Divided by zero exception
Closing the file...
Exiting Method 1
Exiting Run...
Exiting Main...
Press any key to continue_
```

Figure 13. Displaying inner exceptions.

To really understand how this works, you need to put this code into the debugger.

12. Place a breakpoint in Method5 where the exception will be thrown, as shown in Figure 14. Run to the breakpoint by pressing **F5**.



```
            catch (System.ArithmeticException e)
            {
                Console.WriteLine(
                    "Method4 caught an arithmetic exception. Rethrowing...");
                throw; // just toss it back out there
            }
        }

        public void Method5()
        {
            throw new DivideByZeroException(
                "Method5 Divided by zero exception");
        }


        public static void Main()
        {
            Console.WriteLine("In Main()");
            Tester t = new Tester();
            t.Run();
            Console.WriteLine("Exiting Main...");
        }
    }

}
```

```
Build: 1 succeeded, 0 failed, 0 skipped
Deploy: 0 succeeded, 0 failed, 0 skipped
```

Figure 14. Setting the breakpoint.

13. Press **F11** to step into the code. Control transfers to the Catch block in the calling method, Method4. Step in; you'll see the message displayed and the exception is rethrown.

14. The stack is unwound when the exception is rethrown and control transfers to the catch block in the calling method, Method3. This time you create a new exception and throw that.

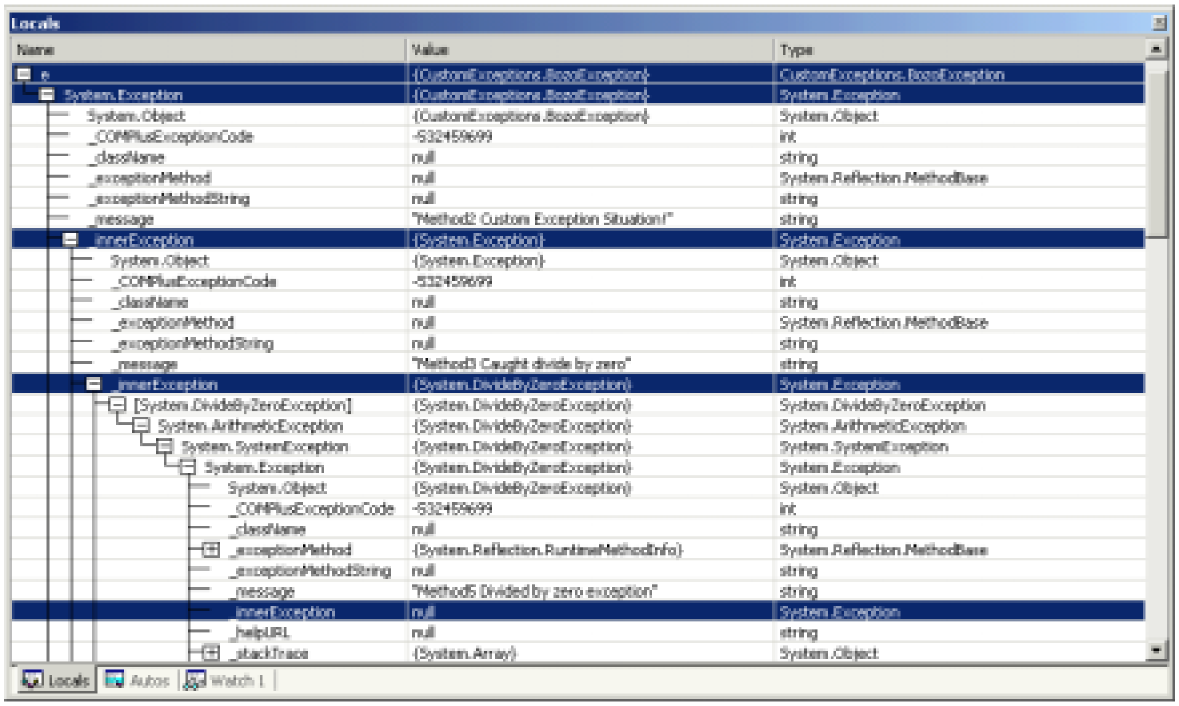
15. Continue to step in; you'll find control passes to the catch block in Method2 where a new exception of type BozoException is created. Continue to press **F11**; you'll step into the constructor for your custom exception, and you'll see the parameters passed to the base class' constructor.

16. Continue stepping back into Method 2; the new BozoException is thrown.

17. The stack continues to unwind and you find yourself back in the catch block of Method1. Before you step further, examine the Locals window. You should see the exception, e.

18. Undock the Locals window to make it nice and big, and expand e, as shown in Figure 15. When you expand e, you'll find System.Exception, the base class. Expand that as well. Inside the base class you'll find many

fields, including _innerException. This is the underlying field supporting the InnerException property.



Locals

Name	Value	Type
e	{CustomExceptions.BozoException}	CustomExceptions.BozoException
System.Exception	{CustomExceptions.BozoException}	System.Exception
System.Object	{CustomExceptions.BozoException}	System.Object
_COMPlusExceptionCode	-532459699	int
_className	null	string
_exceptionMethod	null	System.Reflection.MethodBase
_exceptionMethodString	null	string
_message	"Method2 Custom Exception Situation!"	string
_innerException	{System.Exception}	System.Exception
System.Object	{System.Exception}	System.Object
_COMPlusExceptionCode	-532459699	int
_className	null	string
_exceptionMethod	null	System.Reflection.MethodBase
_exceptionMethodString	null	string
_message	"Method3 Caught divide by zero"	string
_innerException	{System.DivideByZeroException}	System.Exception
[System.DivideByZeroException]	{System.DivideByZeroException}	System.DivideByZeroException
System.ArithmeticException	{System.DivideByZeroException}	System.ArithmeticException
System.SystemException	{System.DivideByZeroException}	System.SystemException
System.Exception	{System.DivideByZeroException}	System.Exception
System.Object	{System.DivideByZeroException}	System.Object
_COMPlusExceptionCode	-532459699	int
_className	null	string
_exceptionMethod	{System.Reflection.RuntimeMethodInfo}	System.Reflection.MethodBase
_exceptionMethodString	null	string
_message	"Method5 Divided by zero exception"	string
_innerException	null	System.Exception
_helpURL	null	string
_stackTrace	{System.Array}	System.Object

Figure 15. Viewing the nested objects.

19. Expand innerException and you'll find the properties of the inner exception object. One of the properties will be _innerException. This is the innerException field of the nested exception.

20. Expand the inner exception object and you'll find a System.DivideByZeroException. Inside you'll find its base class Sytem.ArithemeticException. In that you'll find System.Exception and inside that will be the _innerException field. That innermost innerException field is null, just as you'd expect.

21. Continue stepping through the code. You can watch as inner is assigned to the object referred to by the InnerException property.

22. Each message is displayed in turn until the object inner is null. The loop then finishes, and you'll step into the finally block where the file is closed, and Method 1 unwinds to the Run method, which in turn unwinds to Main, and the program ends.

Summary

- Exceptions are thrown to flag undesirable, but predictable problems in the running of your program.
- Exceptions should not be used to manage bugs or user errors.
- Exceptions are *thrown* and *caught*. The catch block handles the exception and allows your program to continue.
- When an exception is thrown, control moves to the catch block. If there is no appropriate catch block, the stack is unwound until a suitable handler is found.
- A finally block is guaranteed to run whether or not there is an exception.
- A finally block does not require a catch block, but it does require a try block.
- You may derive new exceptions from System.Exception or any other existing exception class.
- Exceptions are presented to their handlers in the order the handlers are written. You must put more specialized handlers before more general handlers.
- The exception type has a number of useful properties, including Message, HelpLink, and StackTrace.
- The Message property allows you to create a custom message. You may pass the message to the constructor of the exception.
- The HelpLink property allows you to provide a URL for help files.
- The StackTrace exception is supported by the CLR and allows you to display or log the line on which the exception is thrown, along with a complete stack trace of the method calls.
- An exception can be rethrown with the keyword *throw.*
- You may nest one exception within another using the InnerException property.

Questions

1. Name three valid circumstances for throwing exceptions and two invalid circumstances.

2. How do you ensure that a resource will be closed (or otherwise disposed of), in the presence of exceptions?

3. How do you pass a custom message to an exception?

4. How do you pass a URL to provide a Help file when an exception is thrown?

5. Can a custom exception derive from Object?

Answers

1. Name three valid circumstances for throwing exceptions and two invalid circumstances.
   A program might throw an exception for any number of reasons, typically when a resource is not available. Three good examples include (1) running out of memory, (2) attempting to open a file and the file is missing or not valid to open, and (3) attempting to contact another machine on the network and your program times out. You do not want to use an exception for (1) a bug in your code or (2) invalid user data entry.

2. How do you ensure that a resource will be closed or otherwise disposed, in the presence of exceptions.
   Be sure to create a try block around any potentially dangerous code, and put the close/dispose code inside a finally block.

3. How do you pass a custom message to an exception?
   You may pass a message to the constructor; it will be assigned to the Message property. The property is read-only, so you must set the message when you construct the object.

4. How do you pass a URL to provide a Help file when an exception is thrown?
   To set the URL of a Help file, use the read/write HelpLink property of the exception.

5. Can a custom exception derive from Object?
   All exceptions must derive from System.Exception, which of course, ultimately derives from Object.

Lab 12: Exceptions

Lab 12 Overview

In this lab you'll learn how to work with exceptions.

To complete this lab, you'll need to work through two exercises:

- Throwing and Catching Exceptions
- Creating Custom Exceptions

Each exercise includes an "Objective" section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

Throwing and Catching Exceptions

Objective

In this exercise, you'll throw and catch exceptions and work with finally blocks.

Things to Consider

- Exceptions are caught in the order most-derived to least derived.
- The finally block is executed whether or not an exception is thrown.
- The .NET Framework provides a number of standard exception classes.
- All Exception classes *must* ultimately derive from System.Exception.

Step-by-Step Instructions

1. Open **Exceptions Starter.sln** in the Exceptions Starter folder.

2. Create a try block and display it to the console.

```
try
{
   Console.WriteLine("Opening file...");
   Console.WriteLine("Calling SecondFunction");
```

3. Call the second function, display it to the console, and end the try block.

```
   SecondFunction();
   Console.WriteLine("Exiting Try...");
}
```

4. Create a catch block to catch DivideByZeroException. Display to the console that DivideByZeroException was caught.

```
catch (System.DivideByZeroException)
{
   Console.WriteLine("Divide by zero exception caught");
}
```

5. Create a generic catch block. Display to the console that an exception was caught.

```
catch
{
   Console.WriteLine("Caught unknown exception...");
}
```

6. Create a finally block. Display to the console that a finally block was caught.

```
finally
{
   Console.WriteLine("Closing the file...");
}
```

7. Implement the SecondFunction. Display the progress to the console.

```
public void SecondFunction()
{
   Console.WriteLine("In SecondFunction");
```

8. Create two double variables and initialize them to small values.

```
double x = 10;
double y = 5;
```

9. Call DoDivide and display the results.

```
Console.WriteLine ("{0} / {1} = {2}", x, y,
                   DoDivide(x,y));
```

10. Set the second variable to zero.

```
y = 0;
```

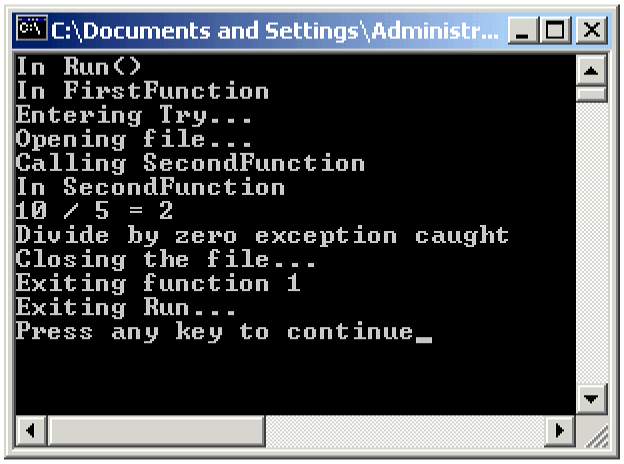
11. Call DoDivide and display the results.

```
Console.WriteLine ("{0} / {1} = {2}", x, y,
                   DoDivide(x,y));
```

12. Implement DoDivide. Test for divide by zero and if found, throw DivideByZeroException.

```
public double DoDivide (double a, double b)
{
   if (b == 0)
      throw new System.DivideByZeroException();
   return a/b;
}
```

13. Compile and run the application as shown in Figure 16.



C:\Documents and Settings\Administr...

```
In Run()
In FirstFunction
Entering Try...
Opening file...
Calling SecondFunction
In SecondFunction
10 / 5 = 2
Divide by zero exception caught
Closing the file...
Exiting function 1
Exiting Run...
Press any key to continue_
```

Figure 16. Final exercise results.

Creating Custom Exceptions

Objective

In this exercise, you'll see how to create and work with your own custom exception class, and how to nest exceptions in multiple throw statements.

Things to Consider

- When you catch an exception and then throw a new exception, you may house the original exception as an inner exception within the new one.
- Exceptions stack one within the other, so that you may retrieve the entire history of inner exceptions.
- Custom exceptions may have methods and members, but it is not uncommon to use an exception object as a signal; its very existence signals the special kind of exception.

Step-by-Step Instructions

1. Open **Custom Exceptions Starter.sln** in the Custom Exceptions Starter folder.

2. Create a custom exception class derived from System.Exception.

```
public class CustomException : System.Exception
{
```

3. Create a constructor that takes two parameters. The first is a string for the exception message. The second is of type Exception and represents the inner exception. Pass both to the base class constructor.

```
public CustomException(
   string message,
   Exception inner):
   base(message, inner)
{ }
```

4. Create a try block and output the progress to the console.

```
try
{
   Console.WriteLine("Opening a file...");
```

5. Call SecondMethod. Output the progress to the console and end the try block.

```
   SecondMethod();
   Console.WriteLine("Exiting Try...");
}
```

6. Catch the CustomException.

```
catch (CustomException e)

{
```

7. Display that you've caught the custom exception and display the exception's message.

```
Console.WriteLine("First Method Caught custom exception");
Console.WriteLine("{0}\n",e.Message);
```

8. Display that you are going to show the exception history.

```
Console.WriteLine("Displaying exception history...");
```

9. Display all the inner exceptions.

```
Exception inner = e.InnerException;
while (inner != null)
{
   Console.WriteLine("{0}", inner.Message);
   inner = inner.InnerException;
}
```

10. Create a finally block. Within the finally block, display that you are closing the file.

```
finally
{
   Console.WriteLine("Closing the file...");
}
```

11. Create a SecondMethod to call the first method.

```
public void SecondMethod()
{
```

12. Make the method call within a try block.

```
try
{
   ThirdMethod();
}
```

13. Catch any Exception object and create a new instance of CustomException, passing in this text as the exception text: “Ave, Caeser, morituri te salutant.” Pass the exception you caught as a second parameter.

```
catch (System.Exception e)
{
   CustomException Custom =
      new CustomException(
      "Ave, Caeser, morituri te salutant.",e);
   throw Custom;
}
```

14. Create the ThirdMethod. This method creates a try block and within it, calls FourthMethod.

```
public void ThirdMethod()
{
   try
   {
      FourthMethod();
   }
```

15. Create a catch block and catch DivideByZeroException. Create a new generic exception and pass in the string "ThirdMethod caught divide by zero" as well as the original exception.

```
catch (System.DivideByZeroException e)
{
   Exception ex =
      new Exception(
      "ThirdMethod caught divide by zero", e);
   throw ex;
}
```

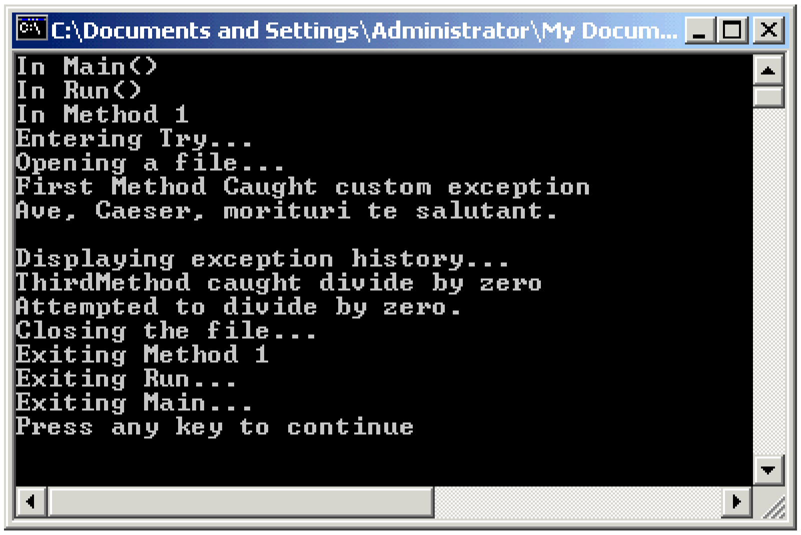
16. Create a FourthMethod.

```
public int FourthMethod()
{
```

17. Create two integer variables. Initialize them to five and zero. Divide the first by the second.

```
int x = 5;
int y = 0;
return x / y;
```

18. Compile and run the application as shown in Figure 17.



```
In Main()
In Run()
In Method 1
Entering Try...
Opening a file...
First Method Caught custom exception
Ave, Caeser, morituri te salutant.

Displaying exception history...
ThirdMethod caught divide by zero
Attempted to divide by zero.
Closing the file...
Exiting Method 1
Exiting Run...
Exiting Main...
Press any key to continue
```

Figure 17. Final exercise results.