throw an exception if you tried to read it after having called `Dispose` on the `StreamReader` object—an obvious coding error—but if there are problems reading the file, `StreamReader` will throw exceptions only from methods or the constructor.

Figure 8-1. Visual Studio reporting an exception

Failures Detected by the Runtime

Another source of exceptions is when the CLR itself detects that some operation has failed. Example 8-2 shows a method in which this could happen. As with Figure 8-1, there's nothing innately wrong with this code (other than not being very useful). It's perfectly possible to use this without causing problems. However, if someone passes in `0` as the second argument, the code will attempt an illegal operation.

Example 8-2. A potential runtime-detected failure

```
static int Divide(int x, int y)
{
    return x / y;
}
```

The CLR will detect when this division operation attempts to divide by zero and will throw a `DivideByZeroException`. This will have the same effect as an exception from an API call: if the program makes no attempt to handle the exception, it will crash, or the debugger will break in.

Division by zero is not always illegal. Floating-point types support special values representing positive and negative infinity, which is what you get when you divide a positive or negative value by zero; if you divide zero by itself, you get the special Not a Number value. None of the integer types support these special values, so integer division by zero is always an error.

The final source of exceptions I described earlier is also the detection of certain failures by the runtime, but they work slightly differently. They are not necessarily triggered directly by anything that your code did on the thread on which the exception occurred. These are sometimes referred to as *asynchronous exceptions*, and in theory they can be thrown at literally any point in your code, making it hard to ensure that you can deal with them correctly. However, these tend to be thrown only in fairly catastrophic circumstances, often when your program is about to be shut down, so only very specialized code needs to deal with these. I will return to them later.

I've described the usual situations in which exceptions are thrown, and you've seen the default behavior, but what if you want your program to do something other than crash?

Handling Exceptions

When an exception is thrown, the CLR looks for code to handle the exception. The default exception handling behavior comes into play only if there are no suitable handlers anywhere on the entire call stack. To provide a handler, we use C#'s `try` and `catch` keywords, as Example 8-3 shows.

Example 8-3. Handling an exception

```
try
{
    using (StreamReader r = new StreamReader(@"C:\Temp\File.txt"))
    {
        while (!r.EndOfStream)
        {
            Console.WriteLine(r.ReadLine());
        }
    }
}
catch (FileNotFoundException)
{
    Console.WriteLine("Couldn't find the file");
}
```

The block immediately following the `try` keyword is usually known as a *try block*, and if the program throws an exception while it's inside such a block, the CLR looks for matching *catch blocks*. Example 8-3 has just a single `catch` block, and in the parentheses following the `catch` keyword, you can see that this particular block is intended to handle exceptions of type `FileNotFoundException`.

You saw earlier that if there is no *C:\Temp\File.txt* file, the `StreamReader` constructor throws a `FileNotFoundException`. In Example 8-1, that caused our program to crash, but because Example 8-3 has a `catch` block for that exception, the CLR will run that `catch` block. At this point, it will consider the exception to have been handled, so the program does not crash. Our `catch` block is free to do whatever it wants, and in this case, my code just displays a message indicating that it couldn't find the file.

Exception handlers do not need to be in the method in which the exception originated. The CLR walks up the stack until it finds a suitable handler. If the failing `StreamReader` constructor call were in some other method that was called from inside the `try` block in Example 8-3, our `catch` block would still run (unless that method provided its own handler for the same exception).