CIS 365 Lab06: Taking Exception

Adam Lewis

May 25, 2018

Contents

1 Objectives

1. Introduce exceptions
2. Understand some of the quirks of exceptions in Java

2 Background

The default behavior in programming languages is to cause your program to terminate when an error happens during execution. This is usually an undesirable situation. Exceptions are a means that we can use to catch when an error occurs in a program and do something to clean up the problem and continue execution.

Exception handling is the process of catching and responding to errors in our programs. In C#, this is achieved through the use of the **try** - **catch** - **finally** construct

2.1 Try blocks

A try block encloses the statements that might throw an exception.

```
try
{
   // statements we think may cause an exception
}
```

2.2 Catch block

```
catch(ExceptionType e)
{
   // statements that process the exception and clean up behind
   // your code
}
```

2.3 Finally block

The finally block is code that you need to always be called regardless of whether or not an exception occurs. This allows you to clean up any unused resources (for example, shutting down a database connection) even if an exception is thrown.

```
finally
{
    // Clean up code that has to be called.
}
```

2.4 The Throws keyword

If you need to explicitly throw an exception, then we can use the **throw** keyword to do this:

```
catch (Exception e)
{
    throw(e);
}
```

2.5 Important points about exceptions

- Exceptions are types that derive directly from **System.Exception** or from one of its descendant classes.
- Exception objects contain detailed information about the error that can be displayed in the **catch** block for debugging purposes
- A try block that can possibly throw multiple exceptions can have more than one catch block to handle those exceptions differently
- Always order catch blocks so that the more specialized catch block comes before a generalized one. This ensures the system processes the specialized catch block first.
- If no exception handler is provided for an exception, the default behavior is that the program terminates with an error message
- By providing a catch block with arguments, we can catch all exceptions that occur inside a try block.

3 Instructions

We will consider a number of different uses of exception handling.

3.1 Basic use of exceptions

Let's start with something simple: start a new Console project in Visual Studio, rename your file and class to **ExceptionTest**, and add the following to the main program:

```
int val = 100;
int div = 0;
int resultVal = (val / div);
Console.WriteLine("The result is " + resultVal);
```

Build and run your application.

3.1.1 Let's do this in a cleaner fashion

The result was pretty spectacular, in a train wreck sort of way. Let's see if we can make this application more user friendly:

```
try {
  int val = 100;
  int div = 0;
  int resultVal = (val / div);
  Console.WriteLine("The result is " + resultVal);
}
catch (System.Exception ex)
{
  Console.WriteLine("Dude, bad show... here's what happened : " + ex.toString()");
}
```

Build and run your application. Not bad... but still not exactly what we would like to see.

3.1.2 Catching multiple exceptions

The **System.Exception** class is the top-level class for all exceptions. So, using it an **catch** block serves as a wild card for all possible exceptions. We would like to be able to do something specific for the divide-by-zero case and then put out a dire warning for all other exceptions. Adjust your code to the following:

```
try {
  int val = 100;
  int div = 0;
  int resultVal = (val / div);
  Console.WriteLine("The result is " + resultVal);
}
catch (System.DivideByZeroException ex)
{
```

```
  Console.WriteLine("Divide by zero? Really?");
  resultVal = 0;
}
catch (System.Exception ex)
{
  Console.WriteLine("Dude, bad show... here's what happened : " + ex.toString()");
  resultVal = 0;
}
```

3.1.3 Using a finally block

This is better but we would like to get some of the regular output out of your program as well. Here's where we could use a **finally**-block. In a more realistic example, one would do things like closing files and databases in this code.

Modify your application to match the following:

```
try {
  int val = 100;
  int div = 0;
  int resultVal = (val / div);
}
catch (System.DivideByZeroException ex)
{
  Console.WriteLine("Divide by zero? Really?");
  resultVal = 0;
}
catch (System.Exception ex)
{
  Console.WriteLine("Dude, bad show... here's what happened : " + ex.toString()");
  resultVal = 0;
}
finally {
  Console.WriteLine("The result is " + resultVal);
}
```

3.2 Dealing with files

The most common use case for exceptions is processing files. Suppose we have an application that is attempting to open the file "c:
unique.txt". Without checking for exceptions, your program terminates on

the opening of the file if it doesn't exist. Start a new application named **ExceptionTest** and modify the **main()** function as follows:

```
File.Open("C:\\unique.txt", FileMode.Open);
```

Try to build and run your application.

3.2.1 Catching IO exceptions

So, we need an exception handler. Change your code as follows:

```
try
{
 File.Open("C:\\unique.txt", FileMode.Open);
}
catch (IOException)
{
   Console.WriteLine("IO");
}
```

Now, build and run your application.

3.2.2 That's better but we can do more

The **IOException** class is a catch-all for all exceptions of this type. We have specialized derived exception classes we can use to check for the more specific cases:

```
try
{
 File.Open("C:\\unique.txt", FileMode.Open);
}
catch (FileNotFoundException)
{
   Console.WriteLine("Not found");
}
catch (IOException)
{

  Console.WriteLine("IO");
}
```

4 Submission instructions

Combine a copy of your source code and a screen-shot of your program into a PDF file (you can use Microsoft Word to do this) and attach it to the assignment on Blackboard. Emacs 24.4.1 (Org mode 8.2.10)