Lecture 19 — Exception Handling

J. Zarnett

jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

March 14, 2015

Acknowledgments: W.D. Bishop

Kinds of Exceptions

An exception is an instance of a detected run-time error.

Exceptions can be generated “automatically” or thrown “manually”.

A “manually” generated exception is when the keyword `throw` appears in the code. A programmer has explicitly `thrown` an exception.

An “automatically” generated exception appears without the appearance of the `throw` keyword. An example is a division by zero.

The line of code is

```
  double result = input / divisor;
```

but an exception will occur here if divisor is zero.

Exceptions are Thrown

In reality, exceptions are always thrown...

Some exceptions are thrown by the system libraries and built-in code, so they appear to be “automatically” generated.

Regardless of the source, handling is the same.

Exception Handling

The process of detecting an exception and attempting to correct the situation is known as exception handling.

Exceptions can be handled or unhandled:

- A handled exception is caught by a portion of the program and corrective action is taken to fix the situation
- An unhandled exception is not caught by a portion of the program and it results in the termination of the program

Undhandled Exception

Here is an example of an unhandled exception:

```
static void Main( )
{
    int i = 0;
    Console.Write( "100 / 0 = " );
    Console.WriteLine ( 100 / i );
}
```

The C# compiler is clever enough to detect an error if we explicitly write (100 / 0) as an expression, so we have to put 0 in a variable `i`.

Handling an Exception

The keyword for handling an exception is `catch`.
(The opposite of `throw`.)

Much like someone who is trying to catch a ball in the air, the `catch` statement needs to be looking in the right direction.

A `catch` statement therefore is explicitly applied to a block of code.

Try-Catch

The syntax for exception handling is the `try-catch` statement.

```
try
{
    // Statement Block
}
catch( ExceptionType e )
{
    // Exception Handling Block
}
```

The keyword `try` indicates what a `catch` is looking at.

The `catch` applies to the statements inside the `try` block.
  It will catch an exception only if it comes from within the `try` block.

ExceptionType

In the syntax for `catch` there is: `ExceptionType e`.

`Exception` is the general form. There are more specific exception types, to allow us to tell different sorts of errors apart.

In addition to saying where we are looking for an exception to be thrown, we also specify what kind of exception we want to `catch`.

`Exception` is general and if we just write `catch ( Exception e )` we will catch most types of exception.

C# Important Exception Types

Some common Exception Types in C#:

Exception Name	**Description**
`Exception`	General exception
`ArgumentException`	Error in an argument passed to a function
`ArithmeticException`	Some type of arithmetic (mathematical) error
`DivideByZeroException`	Division by zero arithmetic error
`ArrayTypeMismatchException`	Type of an array does not match with the type required
`FormatException`	Problem with the formatting of input
`IndexOutOfRangeException`	An array index beyond the bounds of an array
`NullReferenceException`	Use of a null reference

Don’t memorize this list, but you may use it for reference.

Try-Catch Example

Let’s fill in the try-catch example from before.

```
try
{
  double result = input / divisor;
  Console.Write( "Result: ");
  Console.WriteLine( result );
}
catch( DivideByZeroException e )
{
  Console.WriteLine( "Division By Zero Error Detected!" );
}
```

Try-Catch Execution

The only output was the error message.
    The statements to output “Result: ” and the number did not occur.

Like `return` or `break`, when an exception is encountered, execution of the current block is stopped at the point of the exception.

If an exception occurs, no further statements of the `try` block run.

The system will then look for a `catch` block matching the type of the exception that was encountered.

In the example above, there is a matching `catch` block.
    What happens if none is found?

Looking for Someone to Catch

The system will continue looking by going up a level to the function that called the function where the exception was encountered.

```
static void Main ( )
{
    try
    {
      divideNumbers( 100, 0 );
    }
    catch ( DivideByZeroException e )
    {
      Console.WriteLine( "Division By Zero Detected" );
    }
}
```

In the above example, an exception happened in `divideNumbers()` but the `catch` block is found in `Main`.

Looking for Someone to Catch

If the calling function lacks a matching `catch` block, the system goes up another level and repeats this procedure until it gets to `Main`.

If `Main` does not have one either, it's an unhandled exception.
The program terminates with an error message.

In both of the preceding examples, we found a `catch` block matching the type of the exception, so it is handled.

Try-Catch Example

```
try
{
    double result = input / divisor;
    Console.Write( "Result: ");
    Console.WriteLine( result );
}
catch( DivideByZeroException e )
{
  Console.WriteLine( "Division By Zero Error Detected!" );
}
```

So after the exception occurs, control goes to the `catch` block and the statement there executes.

Then the next statement executed is after the end of the `catch` block. Not the console writes.

Variable Scope in Try-Catch

Note that our usual rules of variable scope apply in a `try-catch`.

In the previous example `double result` is only in scope within the `try` block and not after the `try-catch` is done.

Like a function parameter, `DivideByZeroException e` is a parameter available in the `catch` block.

Right now, we won't make use of `e`, but if you are feeling ambitious you can play around with what it contains.

Multiple Types of Exceptions

It is possible to `catch` more than one type of exceptions from a single `try` block.

The syntax and execution for this works like an `if-else` statement.

When an exception is encountered, the exceptions are checked in order from top to bottom, like the `if-else`.

The first `catch` that matches executes.

Only one of the `catch` blocks will execute (mutual exclusivity).

Multiple Types of Exceptions

```
try
{
   complicatedFunction( );
}
catch( DivideByZeroException e )
{
  Console.WriteLine( "Division By Zero Error Detected!" );
}
catch ( ArithmeticException e2 )
{
  Console.WriteLine( "Arithmetic Error Detected!" );
}
catch ( IndexOutOfRangeException e3 )
{
  Console.WriteLine( "Index out of range!" );
}
```

Handling All Exceptions

It’s possible to catch most exceptions using `catch ( Exception e )`

However, there are a few exceptions that will not be caught by this.

To be sure to `catch` all exceptions, use the `catch` statement without the brackets and exception type:

```
try
{
    // Try block
}
catch
{
    // All exceptions caught
}
```

finally

The finally statement provides a mechanism for executing code immediately after the (partial) execution of a try block.

It may be desirable to save the state of the program prior to exiting, regardless of whether an exception has been handled or not.

The finally statement enables a program to exit gracefully, even in the presence of exceptions.

The finally block comes after the catch blocks, if any.

The finally block executes regardless of whether or not an exception is thrown.

finally Syntax

Here is the syntax for the try-catch-finally statement:

```
try
{
    // Statement Block
}
catch( ExceptionType e )
{
    // Exception Handling Block
}
finally
{
    // Finally Block
}
```

`finally` Syntax

Here is an example of the `try-catch-finally` statement:

```
try
{
    Console.WriteLine( "Attempting Division..." );
    divideNumbers(100, 0);
    Console.WriteLine( "Division Succeeded." );
}
catch( ExceptionType e )
{
    Console.WriteLine( "Division by zero detected." );
}
finally
{
    Console.WriteLine( "Operation Complete." );
}
```

finally Syntax

What if we don't have catch?

```
try
{
    Console.WriteLine( "Attempting Division..." );
    divideNumbers(100, 0);
    Console.WriteLine( "Division Succeeded." );
}
finally
{
    Console.WriteLine( "Operation Complete." );
}
```

The finally block runs before the exception terminates the program.

What to Do with Exceptions

Sometimes, when an exception is encountered, the program can take some corrective action and deal with the problem in the `catch` block.

In this lecture we have mostly handled errors by simply reporting them and carrying on.

If the program is a calculator and it takes as input two numbers and an operation, one exception should not end the program.

In other cases, carrying on is harmful and we should stop execution of the program before anything else goes wrong.