

Error Handling in C# with Exceptions

UNDERSTANDING THE IMPORTANCE OF ERROR HANDLING



Jason Roberts

.NET DEVELOPER

@robertsjason dontcodetired.com



Overview




Why handle errors?

Error handling using error codes

Why exceptions?

What is an exception?

Course Outline



Understanding
the Importance
of Error
Handling

Understanding
the Exception
Class Hierarchy

Getting
Started with
Exceptions

Catching,
Throwing, and
Rethrowing
Exceptions

Creating and
Using Custom
Exceptions

Writing
Automated
Tests for
Exception
Throwing Code

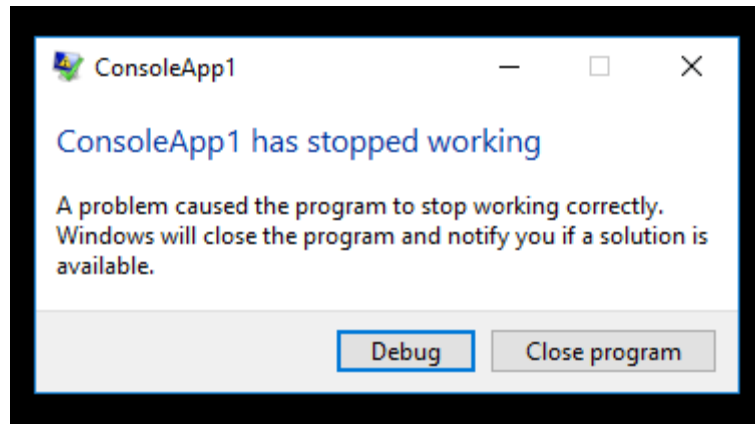


Demo code can be downloaded from
the course page at Pluralsight.com

/before
/after



Why Handle Errors?



Why Handle Errors?



Not crash
program



Chance to
fix/retry



Meaningful
message &
graceful exit



Opportunity to
log error

Good error handling code helps future maintainers understand what possible error conditions may occur and how they can be handled.



Error Handling Using Error Codes

```
private static int ProcessData()  
{  
    // Process some data file  
}
```




```
int errorCode = ProcessData();  
if (errorCode == 0)  
{  
    Console.WriteLine("Processed ok");  
}  
else if (errorCode == 1)  
{  
    Console.WriteLine("Error: Invalid data");  
}  
else if (errorCode == 2)  
{  
    Console.WriteLine("Error: Empty data file");  
}
```



Error Handling Using Error Codes

Need to know all the return values (ints) that represent errors

Need to know all the return values that represent success

Need to remember to add an else if / switch statements for every return value

Program flow will continue as normal even though errors occurred and may cause further damage

May be harder to read than exception handling code

Magic numbers with no semantic meaning harm readability / understanding



```
int errorCode = ProcessData();  
if (errorCode == 0)  
{  
    Console.WriteLine("Processed ok");  
}  
else if (errorCode == 1)  
{  
    Console.WriteLine("Error: Invalid data");  
}  
else if (errorCode == 2)  
{  
    Console.WriteLine("Error: Empty data file");  
}
```



```
int errorCode = ProcessData();  
if (errorCode == OK)  
{  
    Console.WriteLine("Processed ok");  
}  
else if (errorCode == DATA_ERROR)  
{  
    Console.WriteLine("Error: Invalid data");  
}  
else if (errorCode == EMPTY_FILE)  
{  
    Console.WriteLine("Error: Empty data file");  
}
```



Error Handling Using Error Codes

Need to add if / switch statements every time method is called to check return codes

Errors do not “bubble up” the call stack

Catch some errors at a higher level

Catch some errors in a single place

How do you deal with system errors?

- Out of memory
- Access violations

How do you return an error from a constructor?



Why Exceptions?

Don't need to know all error / success codes

Don't need if / switch statements everywhere method is called

More readable, less clutter

No magic numbers / constants

Exceptions can bubble up

Catch exceptions higher up / in one place

Handle system errors

Generate exceptions from constructors



What Is an Exception?

Object

System.Exception

Generated with
the throw
statement

Different exception
classes represent
different errors

Additional error
information

Different exceptions
can be handled
differently



Exception Definitions

Standard
exceptions
provided by the
.NET
Framework

Exceptions
provided by
framework /
library authors

Custom
application
exceptions



Summary



Why handle errors?

- Not crash program
- Chance to fix/retry
- Meaningful message & graceful exit

Error handling using error codes

- if / switch statements
- Magic numbers

Why exceptions?

- More readable, less clutter
- Exceptions can bubble up

What is an exception?

- System.Exception
- .NET Framework, library, custom



Up Next:

Understanding the Exception Class Hierarchy

