

## 13

# Exception Handling



# OBJECTIVES

In this chapter you will learn:

- What exceptions are and how they are handled.
- When to use exception handling.
- To use `try` blocks to delimit code in which exceptions might occur.
- To throw exceptions to indicate a problem.
- To use `catch` blocks to specify exception handlers.
- To use the `finally` block to release resources.
- The .NET exception class hierarchy.
- Exception properties.
- To create user-defined exceptions.



## 13.1 Introduction

- An **exception** is an indication of a problem that occurs during a program's execution.
- Exception handling enables applications to resolve exceptions.
- Exception handling enables clear, **robust** and more **fault-tolerant programs**.



## 13.2 Exception Handling Overview

- Consider the following pseudocode:

*Perform a task*

*If the preceding task did not execute correctly*

*Perform error processing*

*Perform next task*

*If the preceding task did not execute correctly*

*Perform error processing*

...

- In this pseudocode, we begin by performing a task; then we test whether that task executed correctly. If not, we perform error processing.



## 13.2 Exception Handling Overview (Cont.)

- Exception handling enables programmers to remove error-handling code from the “main line” of the program’s execution.
- Programmers can decide to handle all exceptions, all exceptions of a certain type or all exceptions of related types.
- Such flexibility reduces the likelihood that errors will be overlooked.



## Outline

- Figure 13.1's application divides one input integer by a second to obtain an `int` result.

Di vi deByZeroNo  
Excepti onHandl i ng  
.cs

( 1 of 3 )

```
1 // Fig. 13.1: Di vi deByZeroNoExcepti onHandl i ng.cs
2 // Integer division without exception handling.
3 using System;
4
5 class Di vi deByZeroNoExcepti onHandl i ng
6 {
7     static void Mai n()
8     {
9         // get numerator and denominator
10        Console.Wri te( "Please enter an integer numerator: " );
11        int numerator = Convert.ToInt32( Console.ReadLine() );
12        Console.Wri te( "Please enter an integer denominator: " );
13        int denominator = Convert.ToInt32( Console.ReadLine() );
14
15        // divide the two integers, then display the result
```

Converting values can cause a `FormatException`.

Converting values can cause a `FormatException`.

**Fig. 13.1** | Integer division without exception handling. (Part 1 of 3.)



## Outline

```

16     int result = numerator / denominator;
17     Console.WriteLine( "\nResult: {0:D} / {1:D} = {2:D}",
18         numerator, denominator, result );
19 } // end Main
20 } // end class DivideByZeroNoExceptionHandling

```

DivideByZeroNoExceptionHandling.cs

( 2 of 3 )

Division can cause a DivideByZeroException.

```

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14

```

```

Please enter an integer numerator: 100
Please enter an integer denominator: 0

```

**Unhandled Exception: System.DivideByZeroException:**

**Attempted to divide by zero.**

```

at DivideByZeroNoExceptionHandling.Main()
   in C:\examples\ch13\Fig13_01\DivideByZeroNoExceptionHandling\
   DivideByZeroNoExceptionHandling\
   DivideByZeroNoExceptionHandling.cs: line 16

```

**Fig. 13.1** | Integer division without exception handling. (Part 2 of 3.)



## Outline

Di vi deByZeroNo  
Excepti onHandl i ng  
.cs

(3 of 3)

```
Please enter an integer numerator: 100  
Please enter an integer denominator: hello
```

```
Unhandl ed Excepti on: System.FormatExcepti on:
```

```
Input string was not in a correct format.
```

```
at System.Number.StringToNumber(String str, NumberStyles options,  
NumberBuffer& number, NumberFormatInfo info, Boolean parseDecimal)
```

```
at System.Number.ParseInt32(String s, NumberStyles style,  
NumberFormatInfo info)
```

```
at System.Convert.ToInt32(String value)
```

```
at Di vi deByZeroNoExcepti onHandl i ng.Main()
```

```
in C:\exampl es\ch13\Fi g13_01\Di vi deByZeroNoExcepti onHandl i ng\
```

```
Di vi deByZeroNoExcepti onHandl i ng\
```

```
Di vi deByZeroNoExcepti onHandl i ng.cs: l i ne 13
```

**Fig. 13.1** | Integer division without exception handling. (Part 3 of 3.)



## 13.3 Example: Divide by Zero without Exception Handling (Cont.)

- A **FormatException** occurs when `Convert.ToInt32` method receives a string that does not represent a valid integer.
- A program may continue executing even though an exception has occurred and a stack trace has been printed.
- In such cases, the application may produce incorrect results.



## Outline

- This application (Fig. 13.2) uses exception handling to process `DivideByZeroExceptions` and `FormatExceptions`.
- This program demonstrates how to **catch** and **handle** (i.e., deal with) such exceptions.

`DivideByZeroTest`  
`.cs`

( 1 of 4 )

```
1 // Fig. 13.2: DivideByZeroTest.cs
2 // FormatException and DivideByZeroException handlers.
3 using System;
4 using System.Windows.Forms;
5
6 namespace DivideByZeroTest
7 {
8     public partial class DivideByZeroTestForm : Form
9     {
10         public DivideByZeroTestForm()
11         {
12             InitializeComponent();
13         } // end constructor
14
```

**Fig. 13.2** | `FormatException` and `DivideByZeroException` handlers. (Part 1 of 4.)



## Outline

Di vi deByZeroTest  
.CS

( 2 of 4 )

```

15 // obtain 2 integers from the user
16 // and divide numerator by denominator
17 private void divideButton_Click( object sender, EventArgs e )
18 {
19     outputLabel.Text = ""; // clear Label OutputLabel
20
21     // retrieve user input and calculate quotient
22     try
23     {
24         // Convert.ToInt32 generates FormatException
25         // if argument cannot be converted to an integer
26         int numerator = Convert.ToInt32( numeratorTextBox.Text );
27         int denominator = Convert.ToInt32( denominatorTextBox.Text );
28
29         // division generates DivideByZeroException
30         // if denominator is 0
31         int result = numerator / denominator;
32
33         // display result in OutputLabel
34         outputLabel.Text = result.ToString();
35     } // end try

```

**Fig. 13.2** | FormatException and DivideByZeroException handlers. (Part 2 of 4.)



## Outline

### Di vi deByZeroTest .CS

( 3 of 4 )

```

36 catch ( FormatExcepti on )
37 {
38     MessageBox. Show( "You must enter two Integers.",
39         "Invalid Number Format", MessageBoxButtons. OK,
40         MessageBoxI con. Error );
41 } // end catch
42 catch ( Di vi deByZeroExcepti on di vi deByZeroExcepti onParameter )
43 {
44     MessageBox. Show( di vi deByZeroExcepti onParameter. Message,
45         "Attempted to Di vi de by Zero", MessageBoxButtons. OK,
46         MessageBoxI con. Error );
47 } // end catch
48 } // end method di vi deButton_Cl ick
49 } // end class Di vi deByZeroTestForm
50 } // end namespace Di vi deByZeroTest

```

This block catches and handles a FormatExcepti on.

This block catches and handles a Di vi deBy-ZeroExcepti on.

**Fig. 13.2** | FormatExcepti on and Di vi deByZeroExcepti on handlers. (Part 3 of 4.)

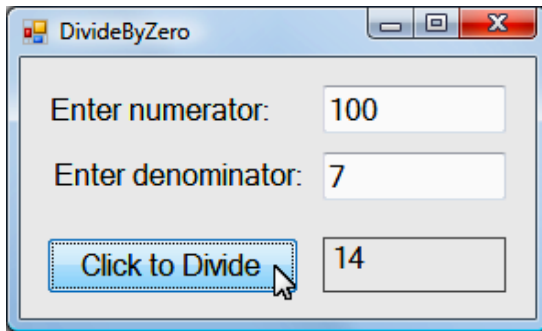


## Outline

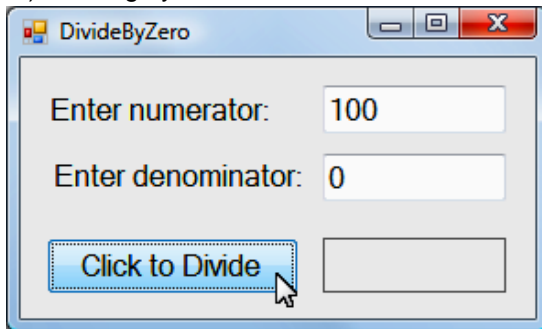
Di vi deByZeroTest  
. CS

( 4 of 4 )

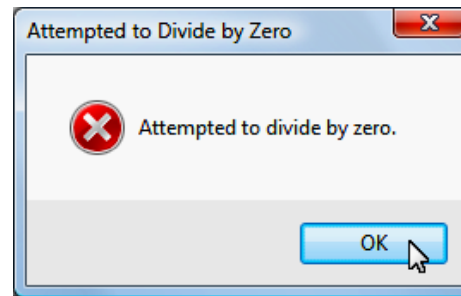
a) Performing integer division



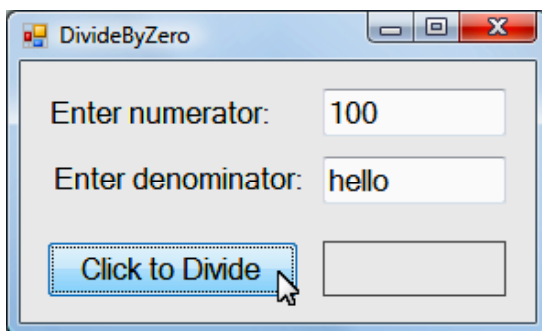
b) Dividing by zero



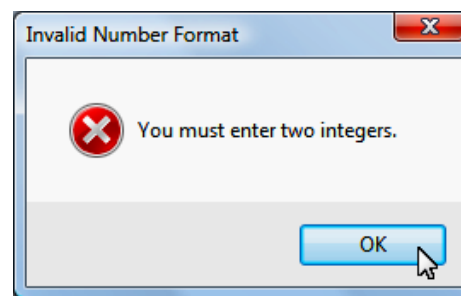
c) Error message



d) Entering a string value



e) Error message



**Fig. 13.2** | FormatExcepti on and Di vi deByZeroExcepti on handlers. (Part 3 of 4.)



## 13.4 Example: Handling `DivideByZeroExceptions` and `FormatExceptions`

- The `Int32.TryParse` method converts a `string` to an `int` value if possible.
- The method requires two arguments—one is the `string` to parse and the other is the variable in which the converted value is to be stored.
- The method returns `true` if the `string` was parsed successfully.
- If the `string` could not be converted, the value `0` is assigned to the second argument.



## 13.4 Example: Handling DivideByZeroExceptions and FormatExceptions (Cont.)

### 13.4.1 Enclosing Code in a try Block

- A **try block** encloses code that might throw exceptions and code that is skipped when an exception occurs.



## 13.4 Example: Handling DivideByZeroExceptions and FormatExceptions (Cont.)

### 13.4.2 Catching Exceptions

- When an exception occurs in a `try` block, a corresponding **catch block** catches the exception and handles it.
- At least one `catch` block must immediately follow a `try` block.
- A `catch` block specifies an exception parameter representing the exception that the `catch` block can handle.
- Optionally, you can include a `catch` block that does not specify an exception type to catch all exception types.



## 13.4 Example: Handling DivideByZeroExceptions and FormatExceptions (Cont.)

### 13.4.3 Uncaught Exceptions

- An **uncaught exception** (or **unhandled exception**) is an exception for which there is no matching **catch** block.



## 13.4 Example: Handling DivideByZeroExceptions and FormatExceptions (Cont.)

- The point at which an exception occurs is called the throw point
- If an exception occurs in a `try` block, program control immediately transfers to the first `catch` block matching the type of the thrown exception.
- After the exception is handled, program control resumes after the last `catch` block.
- This is known as the **termination model of exception handling**.



## 13.5 .NET Exception Hierarchy (Cont.)

- Class **Exception** is the base class of .NET's exception class hierarchy.
- The CLR generates **SystemExceptions**, derived from class **Exception**, which can occur at any point during program execution.
- If a program attempts to access an **out-of-range array index**, the CLR throws an exception of type **IndexOutOfRangeException**.
- Attempting to use a **null** reference causes a **NullReferenceException**.



## 13.5 .NET Exception Hierarchy (Cont.)

- A catch block can use a base-class type to catch a hierarchy of related exceptions.
- A catch block that specifies a parameter of type Exception can catch all exceptions.
- This technique makes sense only if the handling behavior is the same for a base class and all derived classes.

### Common Programming Error 13.3

**The compiler issues an error if a catch block that catches a base-class exception is placed before a catch block for any of that class's derived-class types. In this case, the base-class catch block would catch all base-class and derived-class exceptions, so the derived-class exception handler would never execute—a possible logic error.**

