Chapter 12 – Exception Handling

Outline
12.1  Introduction
12.2  Exception Handling Overview
12.3  Example: Divide by Zero Without Exception Handling
12.4  Example: Handling DivideByZeroExceptions and FormatExceptions
12.5  .NET Exception Hierarchy
12.6  finally Block
12.7  Exception Properties
12.8  User-Defined (Programmer-Defined) Exception Classes

[SKIP: 11.8  Handling Overflows with Operators checked and unchecked]

Many slides modified by Prof. L. Lilien (even many without an explicit message).

Slides added by L.Lilien are © 2006 Leszek T. Lilien.
Permission to use for non-commercial purposes slides added by L.Lilien’s will be gladly granted upon a written (e.g., emailed) request.
12.1 Introduction

• Exception
  – Indication of a problem during program execution
    • Problems are exceptional, normally no problems

• Exception handling
  – Enables programmers to create application that can handle exceptions
    • Enable clear, robust and more fault-tolerant programs

• Two kinds of exception handling:
  – In many cases, exception handling allows to continue ‘correct’ program execution
  – In other more serious cases, exception handling notifies user of a problem and terminates program
12.2 Exception Handling Overview

• Pseudocode with included error processing code

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

...
12.2 Exception Handling Overview

- Including error processing code in a program intermixes program logic with error-handling logic
  - Difficult to read, maintain, debug
  - Better to separate program logic from error-handling logic
    - Errors happen infrequently
    - If many errors, placing checks for errors in program slows it down unnecessarily
      - Unnecessarily because most checks will not find errors

- Exception handling allows to remove error-handling code from the “main line” program execution
  - Improves program clarity
  - Enhances modifiability
12.2 Exception Handling Overview

- **Keywords** for exception handling
  - `try`
    ```
    {
      <block of code in which exceptions might occur>
    }
    ```
  - `catch (<exception_parameter>)`  // 1 or more following each `try`
    ```
    {
      <how to handle this type of exception>
    }
    ```
    Note: If no “(<exception_parameter>)”, catch handles all exception types
  - `finally`  // optional, follows `catch`
    ```
    {
      <codes present here will always execute – whether exception or not>
    }
    ```
12.2 Exception Handling Overview

• **How exception detected?**
  – By a **method** called in a program
  – By **CLR** (Common Language Runtime)

• **What happens when exception detected?**
  – Method or CLR **throws an exception**
    • **Throw point** – point at which exception was thrown
      – Important for debugging

• **Exceptions are objects**
  – Inherit from System.Exceptions
12.2 Exception Handling Overview

- Exception handling scenario
  - Exception occurs in a `try` block
  - The `try` block expires (terminates immediately)
    - Therefore, we say that C# follows the termination model of exception handling
  - CLR searches for the `catch` block (associated with this `try` block) matching the exception
    - Among $n$ `catch` blocks following this `try` block
      - ‘Matching’ – comparing the thrown exception’s type $T_1$ and catch’s exception parameter type $T_2$
        - Match – if they are identical or $T_1$ is a derived class of $T_2$
      - If a match – code within the catch handler is executed
      - Remaining catch blocks are ignored
      - Execution resumed at the first instruction following this `try-catch` sequence
12.2 Exception Handling Overview

- **No exception** in a `try` block => `catch` blocks for this `try` are ignored
  - Execution resumed at the first instruction following this `try-catch` sequence

- **IF** no matching `catch`

  **or:**
  
  **IF** exception occurs in a statement that is **not** in a `try` block

  **THEN:**
  
  - the method containing the statement terminates immediately
  - CLR attempts to locate an enclosing `try` block in a calling method (this is called “Stack unwinding”)

© 2002 Prentice Hall. All rights reserved.  Slide modified by L. Lilien
12.3 Example: Divide by Zero Without Exception Handling

• Read Section 12.3, p. 564.
12.4 Example: Handling *DivideByZeroExceptions* and *FormatExceptions*

- **Error catching** (by a method or CLR)
  - **Method** `Convert.ToInt32` will automatically **detect** invalid representations of an integer
    - Method generates a `FormatException`
  
  - **CLR** automatically **detects** for division by zero
    - CLR generates a `DivideByZeroException`
Program I/O Behavior

When incorrect format are entered into either input fields

When attempting to divide by zero

When attempting to divide by zero.
// Fig 11.1: DivideByZeroTest.cs
// Basics of C# exception handling.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// class demonstrates how to handle exceptions from
// division by zero in integer arithmetic and from
// improper numeric formatting
public class DivideByZeroTest : System.Windows.Forms.Form
{
    private System.Windows.Forms.Label numeratorLabel;
    private System.Windows.Forms.TextBox numeratorTextBox;

    private System.Windows.Forms.Label denominatorLabel;
    private System.Windows.Forms.TextBox denominatorTextBox;

    private System.Windows.Forms.Label outputLabel;

    // required designer variable
    private System.ComponentModel.Container components = null;

    // default constructor
    public DivideByZeroTest()
    {
        // required for Windows Form Designer support
        InitializeComponent();
    }
// main entry point for the application
[STAThread]
static void Main()
{
    Application.Run( new DivideByZeroTest() );
}

// Visual Studio .NET generated code

// obtain integers input by user and divide numerator
// by denominator
private void divideButton_Click(
    object sender, System.EventArgs e )
{
    outputLabel.Text = ""; // clear any prior results
    // retrieve user input and call Quotient
    try
    {
        // Convert.ToInt32 generates FormatException if
        // argument is not an integer
        int numerator = Convert.ToInt32( numeratorTextBox.Text );
        int denominator = Convert.ToInt32( denominatorTextBox.Text );
        int result = numerator / denominator;
        outputLabel.Text = result.ToString();
    } // end try
    // Will not be reached (executed) if an exception is thrown (bec. C# uses termination model of exception handling)
    catch
    { // Try block encloses codes that could result in a throw exception
      // DivideByZeroException thrown by CLR if denominator is zero
        DivideByZeroException thrown by CLR if denominator is zero
      // FormatException thrown by Convert.ToInt32 if it cannot convert string into integer
       FormatException thrown by Convert.ToInt32 if it cannot convert string into integer
      // Will not be reached (executed) if an exception is thrown (bec. C# uses termination model of exception handling)
    }
// process invalid number format
catch (FormatException)
{
    MessageBox.Show("You must enter two integers", "Invalid Number Format", MessageBoxButtons.OK, MessageBoxIcon.Error);
}

// user attempted to divide by zero
catch (DivideByZeroException divideByZeroException)
{
    MessageBox.Show(divideByZeroException.Message, "Attempted to Divide by Zero", MessageBoxButtons.OK, MessageBoxIcon.Error);
}

} // end method divideButton_Click

} // end class DivideByZeroTest
When incorrect format are entered into either input fields

When attempting to diving by zero
12.5 .NET Exception Hierarchy

- .Net Framework exception hierarchy (in the System namespace)
  - Base class Exception
  - Derived class ApplicationException
  - Derived class SystemException
    - Derived class IndexOutOfRangeException
      - Thrown, e.g., on an attempt to access out-of-range array subscript
    - Derived class NullReferenceException
      - Thrown, e.g., on an attempt to reference a non-existing object
    - Derived class ...(many others)...

- In C#, exception-handling mechanisms allows to throw/catch only objects of class Exception and its derived classes
12.5 .NET Exception Hierarchy

- Derived class `ApplicationException`
  - Programmers use it to create exception classes specific to their applications
  - Low chance of program stopping upon ApplicationException
    - Programs can recover from most `ApplicationException`

- Derived class `SystemException`
  - CLR can generate runtime exceptions (derived from `SystemException`) at any point during execution
    - Many can be avoided by proper coding
    - E.g., runtime exception: `IndexOutOfRangeException`
    - E.g., runtime reference to a non-existing object: `NullReferenceException`
  - Typically, program terminates upon `SystemException`
    - Programs can’t recover from most `SystemException`
12.5 .NET Exception Hierarchy

- Is MSDN Library (which provides Help) installed in your Visual Studio?
  - If not, bring 3 CD ROMs to C-208, get and install it

- See full hierarchy of exception classes in .NET in Visual Studio:
  - Select >>Help/Index
  - Look up “Exception class”

- Benefit of exception hierarchy:
  - Instead of catch-ing each derived-class exception class separately, catch the base-class exception
  - Much more concise
  - Useful only if the handling behavior is the same for the base class and the derived classes
12.5 .NET Exception Hierarchy

• How to find out that a program can cause an exception?

1) For methods in .NET Framework
   • Look at detailed method description in Visual Studio
     – Select >> Help > Index
     – Find the class.method in the index
     – Look up the “Exception” section in the document describing the method
       • Enumerates exceptions of the method
       • Describes the reason why each occurs
     • Example (p.574) – `Convert.ToInt32` can throw 2 exception types: `FormatException` and `OverflowException`

2) For CLR

   … next page…
12.5 .NET Exception Hierarchy

- How to find out that a program can cause an exception?

2) For CLR
   - More difficult – search C# Language Specifications (online doc)
   - In Visual Studio
     - Select >> Help>Contents
   - Example:
     - DivideByZeroException described in Section 7.7.2 of language specification
     - Discusses the division operator & when DivideByZeroExceptions occur
12.6 *finally* Block

- **Resource leak**
  - Improper allocation of memory (or other resources)

- **finally** block
  - Associated with a *try* block
  - Ideal for placing resource deallocation code
    - Prevents leak of resources allocated in its *try* block
  - Executed immediately after *catch* handler or *try* block
  - Required if no *catch* block is present
    Optional if one or more *catch* handlers exist
  - Executed if its *try* block entered
    - Even if not completed
12.6 *finally* Block

• The `throw` statement
  – Allows a user/programmer to throw an exception object
    • E.g.:

    ```java
    throw new Exception("Exception in Throw-Test Method");
    ```
    – *string passed* to exception object’s constructor becomes exception object’s error message
  – Can throw only an object of class Exception or one of its derived classes
  – You can customize the exception type thrown from methods
using System;

// demonstrating that ‘finally’ always executes
class UsingExceptions
{
    // entry point for application
    static void Main(string[] args)
    {
        // Case 1: No exceptions occur in called method.
        Console.WriteLine("Calling DoesNotThrowException");
        DoesNotThrowException();

        // Case 2: Exception occurs in called method and is caught
        // in called method.
        Console.WriteLine("\nCalling ThrowExceptionWithCatch");
        ThrowExceptionWithCatch();

        // Case 3: Exception occurs in called method, but not caught
        // in called method, because no catch handlers in called method.
        Console.WriteLine("\nCalling ThrowExceptionWithoutCatch");
        try
        {
            ThrowExceptionWithoutCatch();
        }
    }
}
UsingExceptions.cs

```csharp
// process exception returned from
// ThrowExceptionWithoutCatch
catch
{
    Console.WriteLine("Caught exception from " +
        "ThrowExceptionWithoutCatch in Main");
}

// Case 4: Exception occurs in called method and is caught
// in called method, then rethrown to caller.

// call ThrowExceptionCatchRethrow
try
{
    ThrowExceptionCatchRethrow();
}

// process exception returned from
// ThrowExceptionCatchRethrow
catch
{
    Console.WriteLine("Caught exception from " +
        "ThrowExceptionCatchRethrow in Main");
}

} // end method Main
```

Would process exception that were thrown with no catch handler available

Another static method of class UsingExceptions
public static void DoesNotThrowException()
{
    // try block does not throw any exceptions
    try
    {
        Console.WriteLine( "In DoesNotThrowException" );
    }
    // this catch never executes
    catch
    {
        Console.WriteLine( "This catch never executes" );
    }
    // finally executes because corresponding try executed
    finally
    {
        Console.WriteLine( "Finally executed in DoesNotThrowException" );
    }
    // the following line executed, since normal code execution
    // - without any exceptions
    Console.WriteLine( "End of DoesNotThrowException" );
}

// throws exception and catches it locally
public static void ThrowExceptionWithCatch()
{
    // try block throws exception
    try
    {
        Console.WriteLine( "In ThrowExceptionWithCatch" );
    }
    catch
    {
        Console.WriteLine( "This catch always executes" );
    }
    finally
    {
        Console.WriteLine( "Finally executed in ThrowExceptionWithCatch" );
    }
}
Create a new Exception object

Try block expires because of throw command, program control continue at the first catch following the try block.

Using the exception object’s Message property to access the error message

No catch handlers exist so the program control go directly to the finally block

Try block expires immediately because of “throw new Exception”

Definition for method ThrowExceptionWithoutCatch ()

Try block expires because of throw command, program control continue at the first catch following the try block.
Finally block is reached but program control returns to main immediately after

Program control continue from throw statement to the first catch block that match with the same type

Rethrow the exception back to the calling method for further processing

© 2002 Prentice Hall. All rights reserved.
Finally block reached but program control returns to first occurrence of a try block.
12.6 **finally** Block

- Read subsection on the **using** statement - not to be confused with the using directive for using namespaces
  - p. 581
12.7 Exception Properties

- Caught `Exception` objects have 3 properties (all shown in the next example)

  1) Property `Message`
     
     Cf. Slide 27, line 155: exception object ‘error’
     
     ```csharp
     Console.WriteLine( "Message: " + error.Message );
     
     - Stores the error message associated with an Exception object
       - May be a default message or customized
     
  2) Property `StackTrace`
     
     - Contains a string that represents the method call stack
     
     - Represents sequential list of methods that were not fully processed when the exception occurred
     
     - `throw point` - the exact location where exception occurred
12.7 Exception Properties

• Properties for a caught exception – cont.

3) Property **InnerException**

• Programmers “wrap” exception objects caught in their own code
  – Then can throw new exception types specific to their own code

• The original exception object remains “saved” in **Inner Exception** within the programmer-defined exception

• Example:
  – l. 36 in the following program
using System;

// demonstrates using the Message, StackTrace and // InnerException properties
class Properties
{
    static void Main(string[] args)
    {
        // Call Method1. Any Exception it generates will be // caught in the catch handler that follows
        try
        {
            Method1();
        }
        // Output string representation of Exception, then // output values of InnerException, Message, // and StackTrace properties
        catch (Exception exception)
        {
            Console.WriteLine("exception.ToString(): \n{0}\n", exception.ToString()); // Dump the // entire object named 'exception'
            Console.WriteLine("exception.Message: \n{0}\n", exception.Message); // Dump only the 'Message' property // of the object named 'exception'
            Console.WriteLine("exception.StackTrace: \n{0}\n", exception.StackTrace); // Dump only the 'StackTrace' // property of the object named 'exception'
        }
    }
}

When control returns from stack unwinding, try block is expired sending exception to catch block

Catch block uses method ToString and properties Message, StackTrace and InnerException to produce output
```csharp
Console.WriteLine(
    "exception.InnerException: \n{0}",
    exception.InnerException ); // Dump only the 'InnerExcep-
    // tion' property of the object named 'exception'
} // end catch
} // end Main

// calls Method2
public static void Method1()
{
    Method2();
}

// calls Method3
public static void Method2()
{
    Method3();
}

// throws an Exception containing an InnerException
public static void Method3()
{
    // attempt to convert non-integer string to int
    try
    {
        Convert.ToInt32( "Not an integer" );
    }
}
```

From Method1 control is then returned to the caller which is **Main**

Here also, the CLR *searches for a try block*, but *unsuccessful* it terminates and unwinds from the call stack

Method2 invoked by third on the method stack

Method2 is then unwinds from the method-call stack

When control return to Method3 invoked by Method2 becomes fourth on the method stack

Try block uses `Convert.ToInt32` which become the fifth and final method on stack

Not an integer format, throws a `FormatException`
This removes **Method3** from the method-call stack

Catches the **FormatException** thrown by **Convert.ToInt32**

**Method3** terminates the exception thrown by the **Message** property value

Control will be returned to the statement that invoked **Method2**, which is **Method2**

The next eight lines show the string representation of the **StackTrace** property for the Exception thrown in **Method3**

© 2002 Prentice Hall. All rights reserved.
exception.Message: Exception occurred in Method3

exception.StackTrace:
- at Properties.Method3() in
  f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\properties\properties.cs:line 66
- at Properties.Method2() in
  f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\properties\properties.cs:line 51
- at Properties.Method1() in
  f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\properties\properties.cs:line 45
- at Properties.Main(String[] args) in
  f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\properties\properties.cs:line 16

exception.InnerException: System.FormatException: Input string was not in a correct format.
- at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
- at System.Convert.ToInt32(String s)
- at Properties.Method3() in
  f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_8\properties\properties.cs:line 60

These two line represent the Message property of the exception thrown in Method3

StackTrace property of the exception thrown in Method3

ToString representation of the InnerException property
12.8 User-Defined (Programmer-Defined) Exception Classes

- User can create **customized exception classes** (types)
  - They should **derive from class ApplicationException**
  - Class name should end with “Exception” (e.g., NegativeNumberException)
  - Should define **three constructors**
    - A **default** constructor
    - A constructor that takes a **string** argument
    - A constructor that takes a **string** argument and an **Exception** argument

- Example of customized exception types – next Slide
using System;

// NegativeNumberException represents exceptions caused by illegal
// operations performed on negative numbers

class NegativeNumberException : ApplicationException
{
    // default constructor
    public NegativeNumberException()
    { base( "Illegal operation for a negative number" ) }

    // "string" constructor for customizing error message
    public NegativeNumberException( string message )
    { base( message ) }

    // "string" and "Exception" constructor for customizing error
    // message and specifying inner exception object
    public NegativeNumberException( string message, Exception inner )
    { base( message, inner ) } // end class NegativeNumberException
12.8 User-Defined (Programmer-Defined) Exception Classes

- Next slide: using our customized exception class

- Example test class follows
  - Uses our NegativeNumberException class
  - Expected output
// Fig 11.5: SquareRootTest.cs
// Demonstrating a programmer-defined exception class.

using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

// accepts input and computes the square root of that input
public class SquareRootTest : System.Windows.Forms.Form
{
    private System.Windows.Forms.Label inputLabel;
    private System.Windows.Forms.TextBox inputTextBox;


    private System.Windows.Forms.Label outputLabel;

    // Required designer variable.
    private System.ComponentModel.Container components = null;

    // default constructor
    public SquareRootTest()
    {
        // Required for Windows Form Designer support
        InitializeComponent();
    }

    // Visual Studio .NET generated code
// main entry point for the application
[STAThread]
static void Main()
{
    Application.Run( new SquareRootTest() );
}

// computes the square root of its parameter; throws
// NegativeNumberException if parameter is negative
public double SquareRoot( double operand )
{
    // if negative operand, throw NegativeNumberException
    if ( operand < 0 )
        throw new NegativeNumberException( "Square root of negative number not permitted" );

    // compute the square root
    return Math.Sqrt( operand );
}

// obtain user input, convert to double and calculate
// square root
private void squareRootButton_Click( object sender, System.EventArgs e )
{
    outputLabel.Text = "";

    // catch any NegativeNumberExceptions thrown
    try
    {
        double result =
            SquareRoot( Double.Parse( inputTextBox.Text ) );
    }
outputLabel.Text = result.ToString();

// process invalid number format
catch (FormatException notInteger)
{
    MessageBox.Show(notInteger.Message, "Invalid Operation", MessageBoxButtons.OK, MessageBoxIcon.Error);
}

// display MessageBox if negative number input
try
{
    double result = Math.Sqrt(doubleParseInputValue());
    outputLabel.Text = result.ToString();
}

// end class SquareRootTest

Process the exception caused by FormatException

Catch handler takes care of the NegativeNumberException

Output showing correct function

When attempting to take a negative square root
11.8 Handling Overflows with Operators checked and unchecked

• In .NET, primitive data types are stored in fixed-size structure
  – E.g.: max. for `int` is 2,147,483,647 (32 bits, see p.196)

• Trying to assign a value > max. value causes overflow
  – Overflow causes program to produce incorrect result

• Explicit conversions between integral data types can cause overflow

• C# provides operators `checked` and `unchecked` to specify the validity of integer arithmetic (or to specify a fix for an invalid result)
  – Checked context
    • The CLR throws an `overflowException` if overflow occur during calculation
  – Unchecked context
    • The result of the overflow is truncated
Read these slides on your own-
11.8 Handling Overflows with Operators checked and unchecked

- Use a **checked** context *when* performing calculations that can result in **overflow**
  - Define **exception handlers** to process exception caused by overflow
- Example - below

© 2002 Prentice Hall. All rights reserved.
using System;

class Overflow
{
    static void Main( string[] args )
    {
        int number1 = Int32.MaxValue;  // 2,147,483,647
        int number2 = Int32.MaxValue;  // 2,147,483,647
        int sum = 0;

        Console.WriteLine( "number1: {0}\nnumber2: {1}", number1, number2 );

        // calculate sum of number1 and number2
        try
        {
            Console.WriteLine( "\nSum integers in checked context:" );
            sum = checked( number1 + number2 );
        }
        catch ( OverflowException overflowException )
        {
            Console.WriteLine( overflowException.ToString() );
        }

        Console.WriteLine( "\nsum after checked operation: {0}" , sum );
    }
}
number1: 2147483647
number2: 2147483647

Sum integers in checked context:
System.OverflowException: Arithmetic operation resulted in an overflow.
 at Overflow.Overflow.Main(String[] args) in
 f:\books\2001\csphtp1\csphtp1_examples\ch11\fig11_09\overflow\overflow.cs:line 24

sum after checked operation: 0

Sum integers in unchecked context:
sum after unchecked operation: -2

Addition of number1 and number2 in unchecked context

Sum of the numbers in an unchecked context (bec. the overflowing part is truncated)