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 `DivideByZeroException`s and `FormatException`s
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.
Permision to use for non-commercial purposes slides *added* by L.Lilien's will be gladly granted upon a written (e.g., emailed) request.

© 2002 Prentice Hall. All rights reserved.

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

© 2002 Prentice Hall. All rights reserved.

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

…
```

© 2002 Prentice Hall. All rights reserved.

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 bec. 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

© 2002 Prentice Hall. All rights reserved.

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>
}
```

© 2002 Prentice Hall. All rights reserved.

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

© 2002 Prentice Hall. All rights reserved.

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 T1 and catch's exception parameter type T2
      - Match – if they are identical or T1 is a derived class of T2
  - 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

© 2002 Prentice Hall. All rights reserved.

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.

12.3 Example: Divide by Zero Without Exception Handling

- Read Section 12.3, p. 564.

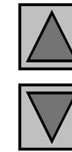
© 2002 Prentice Hall. All rights reserved.

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

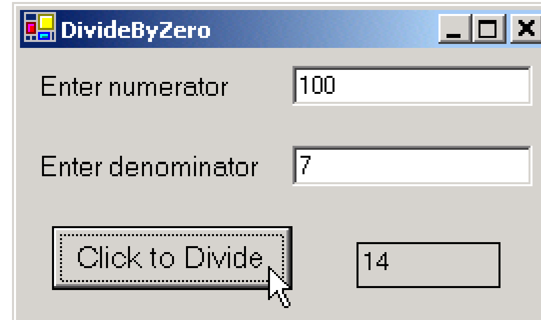
© 2002 Prentice Hall. All rights reserved.

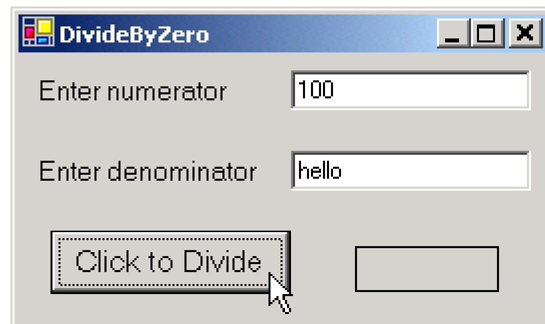
Program I/0 Behavior



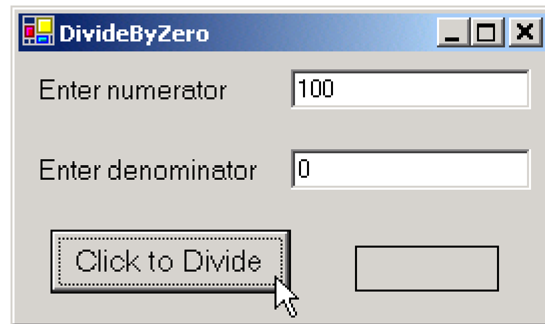
DivideByZeroTest .cs

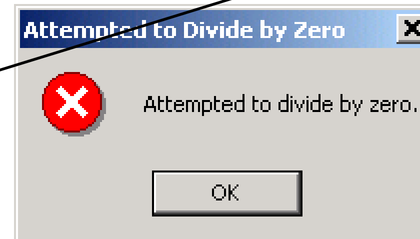
Program Output





When incorrect format are entered into either input fields

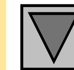




When attempting to divide by zero

Slide copied here by L. Lilien

© 2002 Prentice Hall.
All rights reserved.

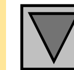


Outline

DivideByZeroTest.cs

```
1    // Fig 11.1: DivideByZeroTest.cs
2    // Basics of C# exception handling.
3
4    using System;
5    using System.Drawing;
6    using System.Collections;
7    using System.ComponentModel;
8    using System.Windows.Forms;
9    using System.Data;
10
11   // class demonstrates how to handle exceptions from
12   // division by zero in integer arithmetic and from
13   // improper numeric formatting
14   public class DivideByZeroTest : System.Windows.Forms.Form
15   {
16      private System.Windows.Forms.Label numeratorLabel;
17      private System.Windows.Forms.TextBox numeratorTextBox;
18
19      private System.Windows.Forms.Label denominatorLabel;
20      private System.Windows.Forms.TextBox denominatorTextBox;
21
22      private System.Windows.Forms.Button divideButton;
23      private System.Windows.Forms.Label outputLabel;
24
25      // required designer variable
26      private System.ComponentModel.Container components = null;
27
28      // default constructor
29      public DivideByZeroTest()
30      {
31         // required for Windows Form Designer support
32         InitializeComponent();
33      }
34
```

© 2002 Prentice Hall.
All rights reserved.



Outline

DivideByZeroTest.cs

```
35      // main entry point for the application
36      [STAThread]
37      static void Main()
38      {
39         Application.Run( new DivideByZeroTest() );
40      }
41
42      // Visual Studio .NET generated code
43
44      // obtain integers input by user and divide numerator
45      // by denominator
46      private void divideButton_Click(
47         object sender, System.EventArgs e )
48      {
49         outputLabel.Text = ""; // clea
50
51         // retrieve user input and cal
52         try
53         {
54            // Convert.ToInt32 generates FormatException if
55            // argument is not an integer
56            int numerator = Convert.ToInt32( numeratorTextBox.Text );
57            int denominator =
58               Convert.ToInt32( denominatorTextBox.Text );
59
60                                            videByZeroException if
61
62                                              denominator;
63
64            outputLabel.Text = result.ToString();
65
66         } // end try
67
```

Try block encloses codes that could result in a throw exception

FormatException thrown by **Convert.ToInt32** if it cannot convert string into integer

DivideByZeroException thrown by CLR if denominator is zero

Will not be reached (executed) if an exception is thrown (bec. C# uses termination model of exception handling)

© 2002 Prentice Hall.
All rights reserved.

Outline

DivideByZeroTest.cs

```
68          // process invalid number format
69          catch ( FormatException )
70          {
71             MessageBox.Show( "You must ente
72                id Number Format",
73                eBoxButtons.OK, MessageBoxIcon.Error );
74          }
75
76          // user attempt
77          catch ( DivideB                          oException )
78          {
79             MessageBox.Show( divideByZeroException.Message,
80                "Attempted to Divide by Zero",
81                MessageBoxButtons.OK, MessageBoxIcon.Error );
82          }
83
84       } // end method divideButton_Click
85
86    } // end class DivideByZeroTest
```

Catch handler for FormatException

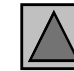
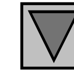
Keyword

Message box to display error message

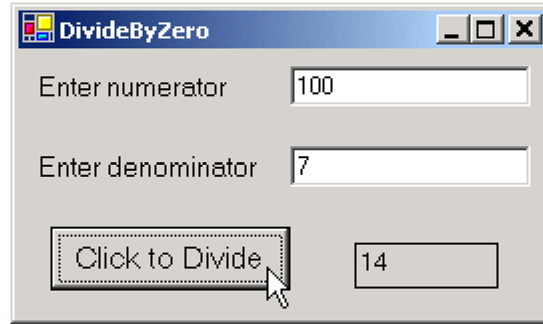
Catch handler for DivideByZeroException

Handler uses property **Message** of class **Exception**

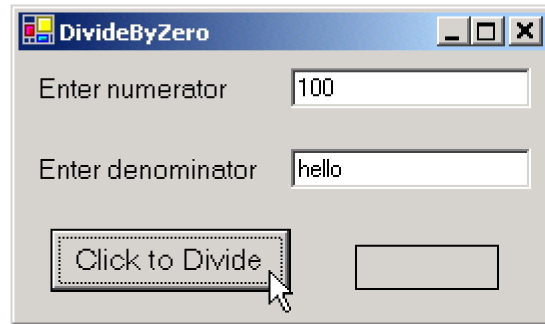
© 2002 Prentice Hall.
All rights reserved.

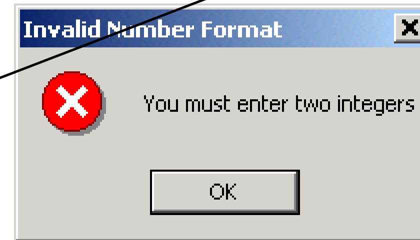



DivideByZeroTest.cs
Program Output

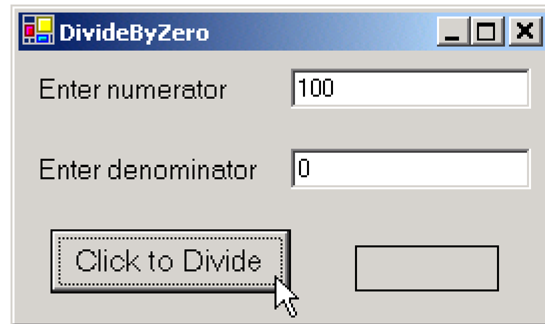


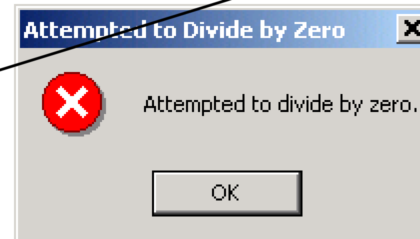
When incorrect format are entered into either input fields





When attempting to diving by zero





© 2002 Prentice Hall.
All rights reserved.

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

© 2002 Prentice Hall. All rights reserved.

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 ApplicationExceptions & continue execution
- 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 SystemExceptions

© 2002 Prentice Hall. All rights reserved.

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

© 2002 Prentice Hall. All rights reserved.

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…

© 2002 Prentice Hall. All rights reserved.

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
  - Expand: Visual Studio .NET, Visual Basic and Visual C#, Reference, Visual C# Language, and C# Language Specification
- Example:
  - **`DivideByZeroException`** described in Section 7.7.2 of language specification
    - Discusses the division operator & when **`DivideByZeroException`**s occur

© 2002 Prentice Hall. All rights reserved.

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 <u>if</u> its **`try`** block entered
    - Even if not completed

© 2002 Prentice Hall. All rights reserved.

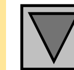
12.6 *finally* Block

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

      ```
      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

© 2002 Prentice Hall. All rights reserved.



UsingExceptions.cs

```
1    // Fig 11.2: UsingExceptions.cs
2    // Using finally blocks.
3
4    using System;
5
6    // demonstrating that 'finally' always executes
7    class UsingExceptions
8    {
9       // entry point for application
10      static void Main( string[] args )
11      {
12         // Case 1: No exceptions occur in called method.
13         Console.WriteLine( "Calling DoesNotThrowException" );
14         DoesNotThrowException();
15
16         // Case 2: Exception occurs in called method and is caught
17         // in called method.
18         Console.WriteLine( "\nCalling ThrowExceptionWithCatch" );
19         ThrowExceptionWithCatch();
20
21      // Case 3: Exception occurs in called method, but not caught
22      // in called method, because no catch handlers in called method.
23               eLine(
24            "\nCalling ThrowExceptionWithoutCatch" );
25
26         // call ThrowExceptionWithoutCatch
27         try
28         {
29            ThrowExceptionWithoutCatch();
30         }
31
```

Static methods of this class so main can invoke directly

Begin try block

© 2002 Prentice Hall.
All rights reserved.

Outline

UsingExceptions.cs

```
32          // process exception returned from
33          // ThrowExceptionWithoutCatch
34          catch
35          {
36             Console.WriteLine( "Caught exception from " +
37                "ThrowExceptionWithoutCatch in Main" );
38          }
39
40                                urs in called method and is caught
41                                           o caller.
42
43                                tionCatchRethrow" );
44
45          // call ThrowExceptionCatchRethrow
46          try
47          {
48             ThrowExceptionCatchRethrow();
49          }
50
51          // process exception returned from
52          // ThrowExceptionCatchRethrow
53          catch
54          {
55             Console.WriteLine( "Caught exception from " +
56                "ThrowExceptionCatchRethrow in Main" );
57          }
58
59       } // end method Main
60
```

Would process exception that were thrown with no catch handler available

atchRethrow

Another static method of class UsingExceptions

© 2002 Prentice Hall.
All rights reserved.

Outline

UsingExceptions.

```
61      // no exceptions thrown
62      public static void DoesNotThrowException()
63      {
64         // try block does not throw any exceptions
65         try
66         {
67            Console.WriteLine( "In DoesNotThrowException" );
68         }
69
70         // this catch never executes
71         catch
72         {
73            Console.WriteLine( "This catch never executes" );
74         }
75
76         // finally executes because corresponding try executed
77         finally
78         {
79            Console.WriteLine(
80               "Finally executed in DoesNotThrowException" );
81         }
82         // the following line executed, since normal code execution
           // - without any exceptions
83         Console.WriteLine( "End of DoesNotThrowException" );
84
85      } // end method DoesNotThrowException
86
87      // throws exception and catches it locally
88      public static void ThrowExceptionWithCatch()
89      {
90         // try block throws exception
91         try
92         {
93            Console.WriteLine( "In ThrowExceptionWithCat
94
```

Definition for method DoesNotThrowException( )

Enters the try block, skips catch block and execute the finally block

End of method, program control returns to Main

Definition for method ThrowExceptionWithCatch( )

© 2002 Prentice Hall.
All rights reserved.

```
95             throw new Exception(
96                "Exception in ThrowExceptionWithCatch" );
97          }     // prepared but did not output error messa
98
99
100
101         {
102            Console.WriteLine( "Message: " + error.Message );
103         }        // message for exception 'error' above is output here
104
105         // finally executes because corresponding t
106         finally
107         {
108            Console.WriteLine(
109               "Finally executed in ThrowExceptionWithCatch" );
110         }
111        // the following code executed bec. exception handled by catch
112         Console.WriteLine( "End of ThrowExceptionWithCatch" );
113
114      } // end method ThrowExceptionWithCatch
115
116      // throws exception and does not catch it locally
117      public static void ThrowExceptionWithoutCatch()
118
119                                                  atch it
120
121
122            Console.WriteLine( "In ThrowExceptionWithoutCatch" );
123
124            throw new Exception(
125               "Exception in ThrowExceptionWithoutCatch" );
126         }           // prepared but did not output error message
127     // No catch in ThrowExceptionWithoutCatch() method. Hence no
        // handling of the exception from l.124. Hence, no printing of
        // this exception's error msg - it is never output.
```

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

Create a new Exception object

ow

ing now becomes the exception object's error message

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

Definition for method ThrowExceptionWithoutCatch ( )

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

Try block expires immediately because of "throw new Exception"

Slide modified by L. Lilien

© 2002 Prentice Hall. All rights reserved.

Outline

ptions.
cs

```
128          // finally executes because corresponding try executed
129          finally
130          {
131             Console.WriteLine( "Finally executed in " +
132                "ThrowExceptionWithoutCatch" );
133          }
134     // Since exception occurred here but was not handled by catch here
        // (bec. no matching catch - actually no catch), CLR handles the
        // exception: (1) executes 'finally', and (2) terminates method
        // terminates method (control returns to Main). So below is
135     // unreachable code (cf. l.82 & 111); would generate logic error
136          Console.WriteLine( "This will never be printed" );
137
138       } // end method ThrowExceptionWithoutCatch
139
140       // throws exception, catches it and rethrows it
141       public static void ThrowExceptionCatchRethrow()
142       {
143          // try block throws exception
144          try
145          {
146             Console.WriteLine( "In ThrowExceptionCatchRethrow"
147
148             throw new Exception(
149                "Exception in ThrowExceptionCatchRethrow" );
150          }
151
152          // catch any exception, place in object error
153          catch ( Exception error )
154          {
155             Console.WriteLine( "Message: " + error.Message );
156                   // catches except
157             // re-throw exceptio
158             throw error;     // Cl
159                              // to Main for further exception handling
160             // unreachable code; would generate logic error
161          }
```

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

Slide modified by L. Lilien

© 2002 Prentice Hall.
All rights reserved.

Outline

tions.

```
162
163        // finally executes because corresponding try executed
164        finally
165        {
166           Console.WriteLine( "Finally executed in " +
167              "ThrowExceptionCatchRethrow" );
168        }
169        // Due to re-throw, CLR passed execution to Main, hence
170        // this is unreachable code (cf. lines 82, 111 & 134)
171        Console.WriteLine( "This will never be printed" );
172
173     } // end method ThrowExceptionCatchRethrow
174
175  } // end class UsingExceptions
```

Finally block reached but program control returns to first occurrence of a try block

Program Output

```
Calling DoesNotThrowException
In DoesNotThrowException
Finally executed in DoesNotThrowException
End of DoesNotThrowException

Calling ThrowExceptionWithCatch
In ThrowExceptionWithCatch
Message: Exception in ThrowExceptionWithCatch
Finally executed in ThrowExceptionWithCatch
End of ThrowExceptionWithCatch

Calling ThrowExceptionWithoutCatch
In ThrowExceptionWithoutCatch
Finally executed in ThrowExceptionWithoutCatch
Caught exception from ThrowExceptionWithoutCatch in Main

Calling ThrowExceptionCatchRethrow
In ThrowExceptionCatchRethrow
Message: Exception in ThrowExceptionCatchRethrow
Finally executed in ThrowExceptionCatchRethrow
Caught exception from ThrowExceptionCatchRethrow in Main
```

Slide modified by L. Lilien

© 2002 Prentice Hall.
All rights reserved.

12.6 *finally* Block

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

© 2002 Prentice Hall. All rights reserved.

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'

     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

© 2002 Prentice Hall. All rights reserved.

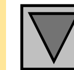
12.7 Exception Properties

- 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

© 2002 Prentice Hall. All rights reserved.



Outline

Properties.cs

```
1     // Fig 11.3: Properties.cs
2     // Stack unwinding and Exception class properties.
3
4     using System;
5
6     // demonstrates using the Message, StackTrace and
7     // InnerException properties
8     class Properties
9     {
10       static void Main( string[] args )
11       {
12          // Call Method1. Any Exception it generates will be
13          // caught in the
14          try
15          {
16             Method1();
17          }
18
19          // Output string representation of Exception, then
20          // output values of InnerException, Message,
21          // and StackTrace properties
22          catch ( Exception exception )
23          {
24             Console.WriteLine(
25                "exception.ToString(): \n{0}\n",
26                exception.ToString() );  // Dump the
27                                         // named 'ex
28             Console.WriteLine( "exception.Message: \n{0}\n",
29                exception.Message ); // Dump only the 'Message' property
30                                     // of the object named 'exception'
31             Console.WriteLine( "exception.StackTrace: \n{0}\n",
32                exception.StackTrace );   // Dump only the 'StackTrace'
33                            // 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

© 2002 Prentice Hall.
All rights reserved.

Outline

Properties.cs

```
34             Console.WriteLine(
35                "exception.InnerException: \n{0}",
36                exception.InnerException ); // Dump only the 'InnerExcep-
37                            // tion' property of the object named 'exception'
38          } // end catch
39
40       } // end Main
41
42       // calls Method2
43       public static void Method1()
44       {
45          Method2();
46       }
47
48       // calls Method3
49       public static void Method2()
50       {
51          Method3();
52       }
53
54       // throws an Exception containing an InnerException
55       public static void Method3()
56       {
57          // attempt to convert non-integer string to int
58          try
59          {
60             Convert.ToInt32( "Not an integer" );
61          }
62
```

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

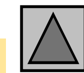
When control return to

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

© 2002 Prentice Hall.
All rights reserved.



Outline

Properties.cs

```
63          // catch FormatException and wrap it in new Exception
64                                 on error )
                                re-throw exce
                                  Method3", error );
71
72     } // end cla
```

This removes **Method3** from the method-call stack

First ... **string** representation of the 'exception' object, returned by method **ToString**

...exception ... call stack

... create ... Exception object, the ...

Method3 termina... exception thrown ... method body

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

Name of the exception class followed by the **Message** property value

Control will be ret... statement that invoked **Method3**, which is **Method2**

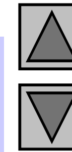
Program Output

```
exception.ToString
System.Exception:
   System.FormatEx                              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
   --- End of inner exception stack trace ---
   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
```

The next eight lines show the **string** representation of the **InnerException** property

Output for the **StackTrace** property for the Exception thrown in **Method3**

© 2002 Prentice Hall.
All rights reserved.



Outline

Properties.cs
Program Output

```
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

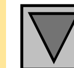
ToString representation of the **InnerException** property

© 2002 Prentice Hall.
All rights reserved.

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

© 2002 Prentice Hall. All rights reserved.



Outline

NegativeNumberException.cs

```
1     // Fig 11:4: NegativeNumberException.cs
2     // NegativeNumberException represents exceptions caused by illegal
3     // operations performed on negative numbers
4
5     using System;
6
7     // NegativeNumberException represents exceptions caused by
8     // illegal operations performed on negative numbers
9     class NegativeNumberException : ApplicationException
          // 'ApplicationException'is base class for user-de
10    {
11       // default constructor
12       public NegativeNumberException()
13          : base( "Illegal operation for a negative number" )
14       {
15       }
16
17       // "string" constructor for customizing error messa
18       public NegativeNumberException( string message )
19          : base( message )
20       {
21       }
22
23       // "string"  and "Exception" constructor for customizing error
24       // message and specifying inner exception object
25       public NegativeNumberException(
26          string message, Exception inner )
27             : base( message, inner )
28       {
29       }
30
31    } // end class NegativeNumberException
```

Class **NegativeNumberException** ... **ption**

This represent the default constructor

This is a constructor that takes in a **string** argument

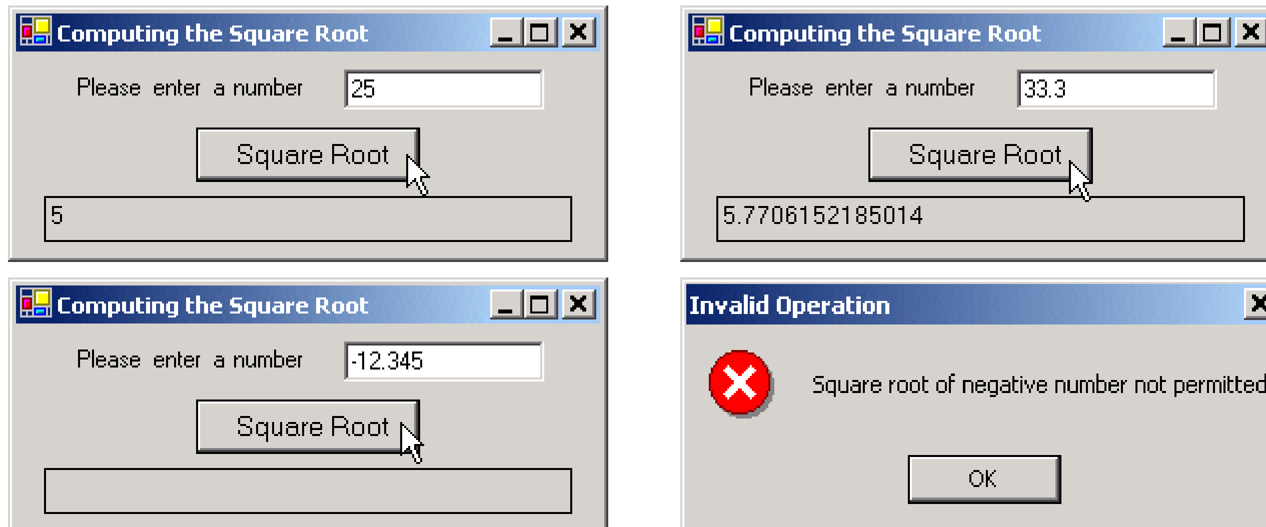
This is a constructor that takes in a string argument and an **Exception** argument

© 2002 Prentice Hall.
All rights reserved.

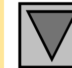
12.8 User-Defined (Programmer-Defined) Exception Classes

- Next slide: using our customized exception class

- Example test class follows
  - Uses our NegativeNumberException class
  - Expected output



© 2002 Prentice Hall. All rights reserved.

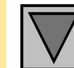


Outline

SquareRootTest.cs

```
1     // Fig 11.5: SquareRootTest.cs
2     // Demonstrating a programmer-defined exception class.
3
4     using System;
5     using System.Drawing;
6     using System.Collections;
7     using System.ComponentModel;
8     using System.Windows.Forms;
9     using System.Data;
10
11    // accepts input and computes the square root of that input
12    public class SquareRootTest : System.Windows.Forms.Form
13    {
14       private System.Windows.Forms.Label inputLabel;
15       private System.Windows.Forms.TextBox inputTextBox;
16
17       private System.Windows.Forms.Button squareRootButton;
18
19       private System.Windows.Forms.Label outputLabel;
20
21       // Required designer variable.
22       private System.ComponentModel.Container components = null;
23
24       // default constructor
25       public SquareRootTest()
26       {
27          // Required for Windows Form Designer support
28          InitializeComponent();
29       }
30
31       // Visual Studio .NET generated code
32
```

© 2002 Prentice Hall.
All rights reserved.



Outline

SquareRootTest.cs

```
33      // main entry point for the application
34      [STAThread]
35      static void Main()
36      {
37         Application.Run( new SquareRootTest() );
38      }
39
40      // computes the square root of its parameter; throws
41      // NegativeNumberException if parameter is negative
42      public double SquareRoot( double operand )
43      {
44         // if negative operand, throw NegativeNumberException
45         if ( operand < 0 )
46            throw new NegativeNumberException(
47               "Square root of negative number not permitted" );
48
49         // compute the square root
50         return Math.Sqrt( operand );
51
52      } // end class SquareRoot
53
54      // obtain user input, convert to double and calculate
55      // square root
56      private void squareRootButton_Click(
57         object sender, System.EventArgs e )
58      {
59         outputLabel.Text = "";
60
61         // catch any NegativeNumberExceptions thrown
62         try
63         {
64            double result =
65               SquareRoot( Double.Parse( inputTextBox.Text ) );
66
```

SqaureRoot throws a **NegativeNumberException**

A **FormatException** occurs if not a valid number from user

Try block invoke **SqaureRoot**

© 2002 Prentice Hall.
All rights reserved.

Outline

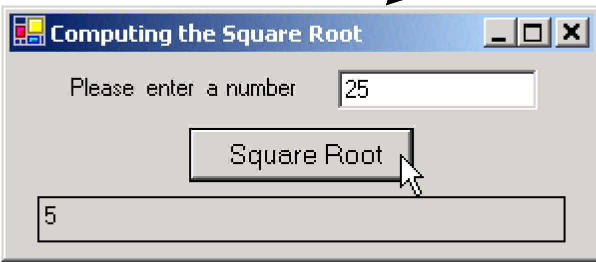
SquareRootTest.cs

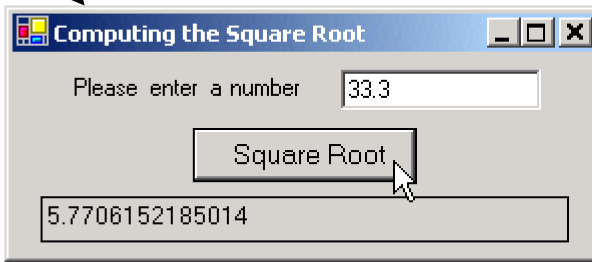
```
67              outputLabel.Text = result.ToString();
68          }
69
70          // process invalid number format
71          catch ( FormatException notInteger )
72          {
73              MessageBox.Show( notInteger.Message,
74                  "Invalid Operation", MessageBoxButtons.OK,
75                  MessageBoxIcon.Error );
76          }
77
78          // display MessageBox if negative number input
79          catch ( NegativeNumberException error )  // cf. sl.37, l.18
80          {
81              MessageBox.Show( error.Message, "Invalid Operation",
82                  MessageBoxButtons.OK, MessageBoxIcon.Error );
83          }
84
85      } // end method squareRootButton_Click
86
87  } // end class SquareRootTest
```

Process the exception caused by **FormatException**

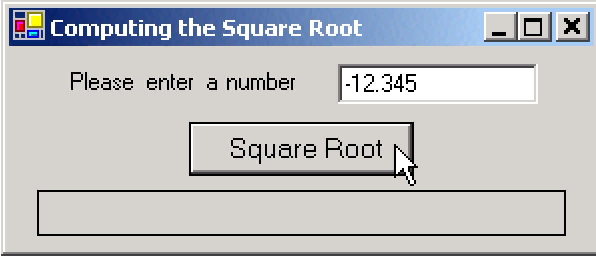
Catch handler takes care of the **NegativeNumberException**

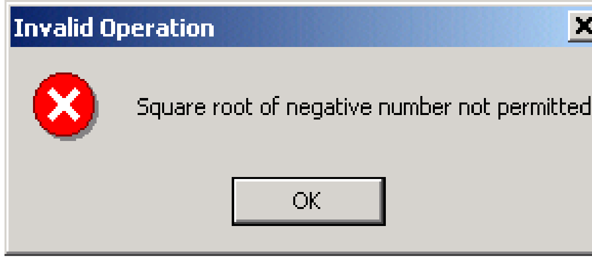
Output showing correct function





When attempting to take a negative square root





© 2002 Prentice Hall.
All rights reserved.

Read these slides on your own-

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

© 2002 Prentice Hall. All rights reserved.

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.

Overflow.cs

```
1    // Fig 11.6: Overflow.cs
2    // Demonstrating operators checked and unchecked.
3
4    using System;
5
6    // demonstrates using the checked and unchecked operators
7    class Overflow
8    {
9       static void Main( string[] args )
10      {
11         int number1 = Int32.MaxValue;  // 2,147,483,647
12         int number2 = Int32.MaxValue;  // 2,147,483,647
13         int sum = 0;
14
15         Console.WriteLine(
16            "number1: {0}\nnumber2: {1}", number1, number2 );
17
18         // calculate sum of number1 and number2
19         try
20         {
21            Console.WriteLine(
22               "\nSum integers in checked context:" );
23
24            sum = checked( number1 + nu
25         }
26
27         // catch overflow exception
28         catch ( OverflowException overflowException )
29         {
30            Console.WriteLine( overflowException.ToString() );
31         }
32
33         Console.WriteLine(
34            "\nsum after checked operation: {0}", sum );
35
```

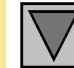
Initialize and declare variables and assigned value to the maximum of int

Number1 and Number2 together causes an overflow, causes **overflowException**

The catch handler gets the **overflowException** and prints out its **string** representation

umber1 and
a checked context

© 2002 Prentice Hall.
All rights reserved.



Outline

Overflow.cs

```
36          Console.WriteLine(
37             "\nSum integers in unchecked context:" );
38
39          sum = unchecked( number1 + number2 );
40
41          Console.WriteLine(
42             "sum after unchecked operation: {0}"
43
44       } // end method Main
45
46    } // end class Overflow
```

Addition of **number1** and **number2** in unchecked context

Program Output

```
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
```

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

© 2002 Prentice Hall.
All rights reserved.