

13

Exception Handling



13.5 .NET Exception Hierarchy (Cont.)

If a method throws exceptions, statements that invoke the method directly or indirectly should be placed in `try` blocks, and those exceptions should be caught and handled.

- Search for “`Convert.ToInt32` method” in the **Index** of the Visual Studio online documentation.
- Select the document entitled **Convert.ToInt32 Method (System)**.
- In the document that describes the method, click the link **ToInt32(String)**.
- The **Exceptions** section indicates that method `Convert.ToInt32` throws two exception types.



13.6 finally Block (Cont.)

- C# provides the `finally` block, which is guaranteed to execute regardless of whether an exception occurs.
- This makes the `finally` block ideal to release resources from the corresponding `try` block.
- A `finally` block typically contains code to release resources acquired in the corresponding `try` block, which makes the `finally` block an effective mechanism for eliminating resource leaks.



Outline

- The application in Fig. 13.4 demonstrates that the `finally` block always executes.

UsingExceptions.cs

(1 of 10)

```

1 // Fig. 13.4: UsingExceptions.cs
2 // Using finally blocks.
3 // finally blocks always execute, even when no exception occurs.
4 using System;
5
6 class UsingExceptions
7 {
8     static void Main()
9     {
10         // Case 1: No exceptions occur in called method
11         Console.WriteLine( "Calling DoesNotThrowException" );
12         DoesNotThrowException();
13
14         // Case 2: Exception occurs and is caught in called method
15         Console.WriteLine( "\nCalling ThrowExceptionWithCatch" );
16         ThrowExceptionWithCatch();
17
18         // Case 3: Exception occurs, but is not caught in called method
19         // because there is no catch block.
20         Console.WriteLine( "\nCalling ThrowExceptionWithoutCatch" );

```

Main invokes method
DoesNotThrowException.

Main invokes method
ThrowExceptionWithCatch.

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 1 of 8.)



Outline

UsingExceptions.cs

(2 of 10)

```
21
22 // call ThrowExceptionWithoutCatch
23 try
24 {
25     ThrowExceptionWithoutCatch();
26 } // end try
27 catch
28 {
29     Console.WriteLine( "Caught exception from " +
30         "ThrowExceptionWithoutCatch in Main" );
31 } // end catch
32
33 // Case 4: Exception occurs and is caught in called method,
34 // then rethrown to caller.
35 Console.WriteLine( "\nCalling ThrowExceptionCatchRethrow" );
36
37 // call ThrowExceptionCatchRethrow
38 try
39 {
40     ThrowExceptionCatchRethrow();
41 } // end try
```

Fig. 13.4 | finally blocks always execute, even when no exception occurs. (Part 2 of 8.)



Outline

UsingExceptions.cs

(3 of 10)

```
42     catch
43     {
44         Console.WriteLine( "Caught exception from " +
45             "ThrowExceptionCatchRethrow in Main" );
46     } // end catch
47 } // end method Main
48
49 // no exceptions thrown
50 static void DoesNotThrowException()
51 {
52     // try block does not throw any exceptions
53     try
54     {
55         Console.WriteLine( "In DoesNotThrowException" );
56     } // end try
57     catch
58     {
59         Console.WriteLine( "This catch never executes" );
60     } // end catch
```

Because the `try` block does not throw any exceptions, the `catch` block is ignored.

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 3 of 8.)



Outline

```

61  finally
62  {
63      Console.WriteLine( "finally executed in DoesNotThrowException" );
64  } // end finally
65
66      Console.WriteLine( "End of DoesNotThrowException" );
67  } // end method DoesNotThrowException
68
69  // throws exception and catches it locally
70  static void ThrowExceptionWithCatch()
71  {
72      // try block throws exception
73      try
74      {
75          Console.WriteLine( "In ThrowExceptionWithCatch" );
76          throw new Exception( "Exception in ThrowExceptionWithCatch" );
77      } // end try
78      catch ( Exception exceptionParameter )
79      {
80          Console.WriteLine( "Message: " + exceptionParameter.Message );
81      } // end catch

```

UsingExceptions.cs

(4 of 10)

The finally block
always executes.

The try block
throws an exception.

The catch and finally
blocks execute when the
exception occurs.

Fig. 13.4 | finally blocks always execute, even when no exception occurs. (Part 4 of 8.)



Outline

UsingExceptions.cs

(5 of 10)

```
82     finally
83     {
84         Console.WriteLine(
85             "finally executed in ThrowExceptionWithCatch" );
86     } // end finally
87
88     Console.WriteLine( "End of ThrowExceptionWithCatch" );
89 } // end method ThrowExceptionWithCatch
90
91 // throws exception and does not catch it locally
92 static void ThrowExceptionWithoutCatch()
93 {
94     // throw exception, but do not catch it
95     try
96     {
97         Console.WriteLine( "In ThrowExceptionWithoutCatch" );
98         throw new Exception( "Exception in ThrowExceptionWithoutCatch" );
99     } // end try
```

The catch and finally blocks execute when the exception occurs.

The try block throws an exception.

Fig. 13.4 | finally blocks always execute, even when no exception occurs. (Part 5 of 8.)



Outline

UsingExceptions.cs

(6 of 10)

```
100     finally
101     {
102         Console.WriteLine( "finally executed in " +
103             "ThrowExceptionWithoutCatch" );
104     } // end finally
105
106     // unreachable code; logic error
107     Console.WriteLine( "End of ThrowExceptionWithoutCatch" );
108 } // end method ThrowExceptionWithoutCatch
109
110 // throws exception, catches it and rethrows it
111 static void ThrowExceptionCatchRethrow()
112 {
113     // try block throws exception
114     try
115     {
116         Console.WriteLine( "In ThrowExceptionCatchRethrow" );
117         throw new Exception( "Exception in ThrowExceptionCatchRethrow" );
118     } // end try
```

The finally block executes but the exception remains uncaught until after control returns to Main.

The try block throws an exception.

Fig. 13.4 | finally blocks always execute, even when no exception occurs. (Part 6 of 8.)



Outline

UsingExceptions.cs

(7 of 10)

```

119 catch ( Exception exceptionParameter )
120 {
121     Console.WriteLine( "Message: " + exceptionParameter.Message );
122
123     // rethrow exception for further processing
124     throw;
125
126     // unreachable code; logic error
127 } // end catch
128 finally
129 {
130     Console.WriteLine( "finally executed in " +
131         "ThrowExceptionCatchRethrow" );
132 } // end finally
133
134 // any code placed here is never reached
135 Console.WriteLine( "End of ThrowExceptionCatchRethrow" );
136 } // end method ThrowExceptionCatchRethrow
137 } // end class UsingExceptions

```

The catch block rethrows the exception, which is then caught after control returns to Main.

The catch and finally blocks execute when the exception occurs.

Fig. 13.4 | finally blocks always execute, even when no exception occurs. (Part 7 of 8.)



```
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
```

Fig. 13.4 | `finally` blocks always execute, even when no exception occurs. (Part 8 of 8.)



13.7 Exception Properties

- Class `Exception`'s properties are used to formulate error messages indicating a caught exception.
 - Property `Message` stores the error message associated with an `Exception` object.
 - Property `StackTrace` contains a `string` that represents the **method-call stack**.



13.8 User-Defined Exception Classes

- In some cases, you might create exception classes specific to the problems that occur in your programs.
- **User-defined exception classes** should derive directly or indirectly from class `Exception` of namespace `System`.
- Exceptions should be documented so that other developers will know how to handle them.



13.8 User-Defined Exception Classes (Cont.)

- User-defined exceptions should define three constructors:
 - a parameterless constructor
 - a constructor that receives a `string` argument (the error message)
 - a constructor that receives a `string` argument and an `Exception` argument (the error message and the inner exception object)



Outline

- Class `NegativeNumberException` (Fig. 13.6) represents exceptions that occur when a program performs an illegal operation on a negative number.

NegativeNumberException.cs

(1 of 2)

```

1 // Fig. 13.6: NegativeNumberException.cs
2 // NegativeNumberException represents exceptions caused by
3 // illegal operations performed on negative numbers.
4 using System;
5
6 namespace SquareRootTest
7 {
8     class NegativeNumberException : Exception
9     {
10         // default constructor
11         public NegativeNumberException()
12             : base( "Illegal operation for a negative number" )
13         {
14             // empty body
15         } // end default constructor
16

```

Inheriting from class
Exception.

Parameterless constructor.

Fig. 13.6 | `NegativeNumberException` represents exceptions caused by illegal operations performed on negative numbers. (Part 1 of 2.)



Outline

NegativeNumberException.cs

(2 of 2)

```

17 // constructor for customizing error message
18 public NegativeNumberException( string messageValue )
19     : base( messageValue )
20 {
21     // empty body
22 } // end one-argument constructor
23
24 // constructor for customizing the exception's error
25 // message and specifying the InnerException object
26 public NegativeNumberException( string messageValue,
27     Exception inner )
28     : base( messageValue, inner )
29 {
30     // empty body
31 } // end two-argument constructor
32 } // end class NegativeNumberException
33 } // end namespace SquareRootTest

```

Constructor with a single argument (the Message).

Constructor with two arguments (the Message and InnerException).

Fig. 13.6 | NegativeNumberException represents exceptions caused by illegal operations performed on negative numbers. (Part 2 of 2.)



Outline

- Class `SquareRootForm` (Fig. 13.7) demonstrates our user-defined exception class.

`SquareRootTest.cs`

(1 of 4)

```
1 // Fig. 13.7: SquareRootTest.cs
2 // Demonstrating a user-defined exception class.
3 using System;
4 using System.Windows.Forms;
5
6 namespace SquareRootTest
7 {
8     public partial class SquareRootForm : Form
9     {
10         public SquareRootForm()
11         {
12             InitializeComponent();
13         } // end constructor
14
```

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 1 of 4.)



Outline

```

15 // computes square root of parameter; throws
16 // NegativeNumberException if parameter is negative
17 public double SquareRoot( double value )
18 {
19     // if negative operand, throw NegativeNumberException
20     if ( value < 0 )
21         throw new NegativeNumberException(
22             "Square root of negative number not permitted" );
23     else
24         return Math.Sqrt( value ); // compute square root
25 } // end method SquareRoot

27 // obtain user input, convert to double, calculate square root
28 private void squareRootButton_Click( object sender, EventArgs e )
29 {
30     outputLabel.Text = ""; // clear outputLabel
31
32     // catch any NegativeNumberException thrown
33     try
34     {
35         double result =
36             SquareRoot( Convert.ToDouble( inputTextBox.Text ) );
37

```

SquareRootTest.cs

(2 of 4)

If the numeric value that the user enters is negative, SquareRoot throws a NegativeNumberException.

SquareRoot invokes Math's Sqrt method.

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 2 of 4.)



Outline

SquareRootTest.cs

(3 of 4)

```
38     outputLabel.Text = result.ToString();
39 } // end try
40 catch ( FormatException formatExceptionParameter )
41 {
42     MessageBox.Show( formatExceptionParameter.Message,
43         "Invalid Number Format", MessageBoxButtons.OK,
44         MessageBoxIcon.Error );
45 } // end catch
46 catch ( NegativeNumberException
47     negativeNumberExceptionParameter )
48 {
49     MessageBox.Show( negativeNumberExceptionParameter.Message,
50         "Invalid Operation", MessageBoxButtons.OK,
51         MessageBoxIcon.Error );
52 } // end catch
53 } // end method squareRootButton_Click
54 } // end class SquareRootForm
55 } // end namespace SquareRootTest
```

Catching and handling a
NegativeNumber-
Exception.

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 3 of 4.)

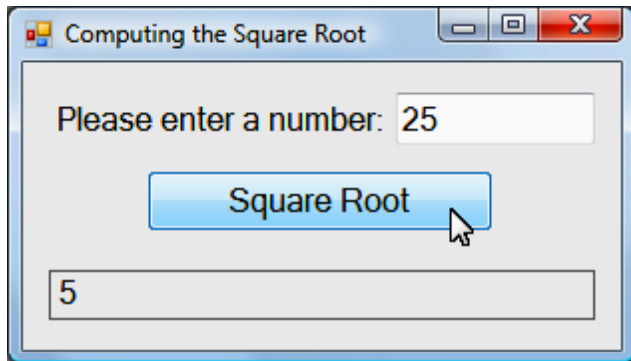


Outline

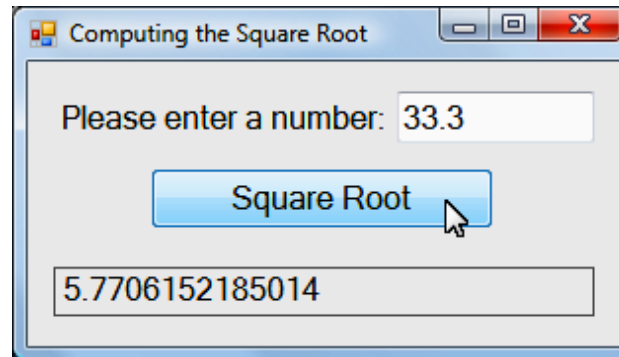
SquareRootTest.cs

(4 of 4)

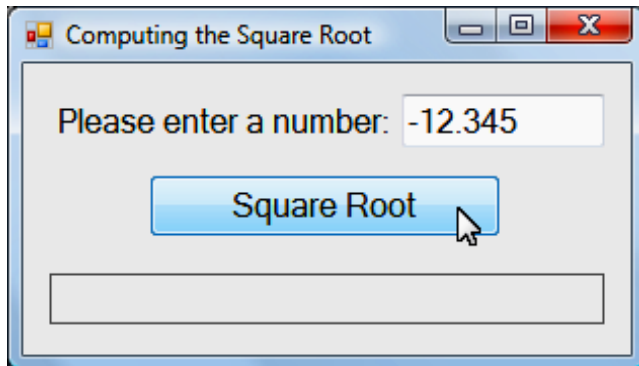
a) Calculating an integer square root



b) Calculating a double square root



c) Attempting a negative square root



d) Error message displayed

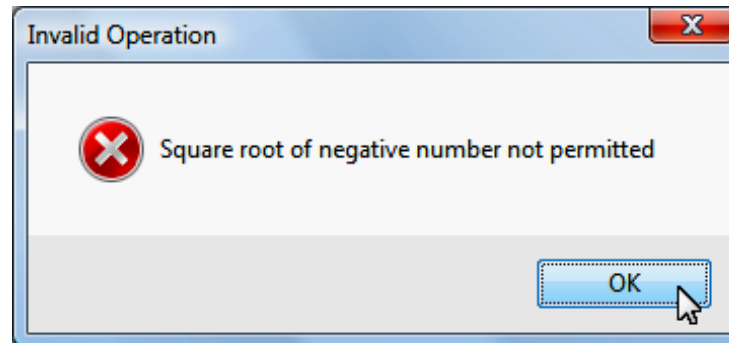


Fig. 13.7 | Demonstrating a user-defined exception class. (Part 4 of 4.)

