Programming with Event Driven in C#

Exception Handling and Data Validation

Chalew Tesfaye
Department of Computer Science
Debre Berhan University
2017

Objectives

- After the end this lesson the student will be able to
  - ✓ Debug application
  - ✓ Understand exception in C#
  - ✓ Write a program that Handle exception
  - ✓ Write a program that Validate data

Lesson Outline

- Debug application
- Exception Overview
- .NET Exception Hierarchy
- Handling Exception
- Throwing exception
- Programmer-Defined Exception

Debug Application

Discover how the Debugger can be used to find run-time errors

Errors

- Visual Studio IDE reports errors as soon as it is able to detect a problem
- Syntax errors
  - ✓ Language rule violation
- Run-Time Errors
  - ✓ Just because your program reports no syntax errors does not necessarily mean it is running correctly
  - ✓ One form of run-time error is a logic error
    - > Program runs, but produces incorrect results
    - > May be off-by-one in a loop
    - > Sometime users enter incorrect values
  - ✓ Finding the problem can be challenging
- Logical error
  - ✓ Missing business logic

Debugging Application

- Debugging is the most difficult and time consuming phase of programing.
- Visual studio offers many tools for testing and debugging.
- Before you begin debugging, you can set the options that control how Visual Studio handles exceptions.
  - ✓ **Set debugging options**
    - > Use **Debug->Option and Setting** menu to open Option dialog box
- Debuggers let you observer the run-time behavior
  - ✓ break or halt execution
  - ✓ step through the application
  - ✓ evaluate variables
  - ✓ set breakpoints
- Debug menu offers debugging options

Exception Handling

Dealing with is a problem that arises during the execution of a program.

Exception

- An exception is an indication of a problem that occurs during a program's execution
- Exception handling enables programmers to create applications that can resolve (or handle) exceptions
- In many cases, handling an exception allows a program to continue executing as if no problem was encountered.
- A more severe problem may prevent a program from continuing normal execution
  - ✓ instead requiring the program to notify the user of the problem,
  - ✓ then terminate in a controlled manner.

Exception Handling

- Exception handling
  - ✓ Enable clear, robust and more fault-tolerant programs
  - ✓ Process synchronous errors
  - ✓ Follows the termination model of exception handling
  - ✓ Enable to build fault-tolerant programs
  - ✓ Provided by **`System.Exception`**
- Keywords
  - ✓ `try`
    - > Include code in which exceptions might occur
  - ✓ `catch`
    - > Code to handle the exception
    - > Must be of class Exception or one that extends it directly or indirectly
  - ✓ `finally`
    - > (Optional) code present here will always execute

Finally Block

- Resource leak
  - ✓ Aborting a program and leaving a resource in a state in which other programs are unable to acquire the resource
- Finally block
  - ✓ Ideal for deallocation code to release resources acquired in **try** block
  - ✓ Execute immediately after catch handler or try block
  - ✓ Must be present if no catch block is present
  - ✓ Is optional if more than one or more catch handler exist

Exception handling - Syntax

```
try {
        //statement
}
catch(Exception)
{
    //exception statement, executed if error occurred
}
finally {
        //code to release resources
}
```

When to use exception handling

- Avoid exception handling except for error handling
- Try to include from inception of design
- Exception has little overhead, so handling more efficient than trying to perform error handling with if statements.
  - ✓ Use only for infrequent problems
- Methods with common error conditions should return **null** rather than throwing exceptions.
- Parameter-less **catch** handler must be last one

.NET Exception Hierarchy

- .Net Framework
  - ✓ Class `Exception` is base class
  - ✓ Derived classes:
    - > `ApplicationException`
      - Programmer use to create data types specific to their application
      - Low chance of program stopping execution
      - Can create programmer-defined exception classes
    - > `SystemException`
      - CLR can generate at any point during execution
      - Runtime exception
      - Can be avoided with proper coding
        - Example: IndexOutOfRangeException, FormatException, …

Library Exceptions

- Feel free to use these:
  - ✓ ArithmeticException
  - ✓ ArrayTypeMismatchException
  - ✓ DivideByZeroException
  - ✓ IndexOutOfRangeException
  - ✓ InvalidCastException
  - ✓ NullReferenceException
  - ✓ OutOfMemoryException
  - ✓ OverflowException
  - ✓ StackOverflowException
  - ✓ TypeInitializationException
  - ✓ …

Exception hierarchy > Example

```
try    {
    mark = Convert.ToDouble(txtTotalMark.Text);
    lg = crs.getLetterGrade(mark);
    //calculate grade point of a course
    credit = Convert.ToInt32(txtCredit.Text);
    lgp = crs.getletterGradePoint();
    gp = crs.getGradePoint(credit, lgp);
    //display the result
    txtLetterGrade.Text = lg.ToString();
    txtGradePoint.Text = gp.ToString();
  }
  catch(OverflowException)   {    MessageBox.Show("Over flow exception.");         }
  catch (FormatException)     {   MessageBox.Show("Format Exception Occured."); }
  catch (Exception) {   MessageBox.Show("General Exception.");     }
```

Throw exception

- Programmer can detect non-system errors and cause an exception to be thrown
- use **throw** (an exception object)
- Must be of either class Exception or one of its derived classes
- Customize the exception type thrown from methods

Throw exception > Example

```
try
 {
    mark = Convert.ToDouble(txtTotalMark.Text);
    lg = crs.getLetterGrade(mark);
    //calculate grade point of a course
    credit = Convert.ToInt32(txtCredit.Text);
    if(credit < 0)
    {
       throw new Exception("Credit hour can not be negative Value.");
    }
    lgp = crs.getletterGradePoint();
    gp = crs.getGradePoint(credit, lgp);
    //display the result
    txtLetterGrade.Text = lg.ToString();
    txtGradePoint.Text = gp.ToString();
 }
 catch(Exception ex)
 {
    MessageBox.Show(ex.Message);
 }
```

Exception Properties

- Message
  - Stores the error message associated with an Exception object
    - May be a default message or customized
- StackTrace
  - Contain a string that represents the *method call stack*
  - Represent sequential list of methods that were not fully processed when the exception occurred
  - The exact location is called the *throw point*
- *GetType*
  - ✓ *Gets the type of the current exception*
- InnerException
  - ✓ "Wrap" exception objects caught in code, then throw new exception types

Exception Properties > Example

```
try
  {
     mark = Convert.ToDouble(txtTotalMark.Text);
     lg = crs.getLetterGrade(mark);
     //calculate grade point of a course
     credit = Convert.ToInt32(txtCredit.Text);
     lgp = crs.getletterGradePoint();
     gp = crs.getGradePoint(credit, lgp);
     //display the result
     txtLetterGrade.Text = lg.ToString();
     txtGradePoint.Text = gp.ToString();
  }
  catch (Exception ex)
  {
     MessageBox.Show(ex.Message +"\n"+ex.StackTrace.ToString() +"\n"+ex.GetType() );
  }
```

Programmer-Defined Exception Classes

- Creating customized exception types
  - ✓ Should derive from class ApplicationException
  - ✓ Should end with "Exception"
  - ✓ Should define three constructors
    - A default constructor
    - A constructor that receives a string argument
    - A constructor that takes a string argument and an Exception argument

Programmer-Defined Exception > Example

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

        }
        // constructor for customizing error message
        Public NegativeNumberException( stringmessage ) : base( message )
        {

        }
        // constructor for customizing error message and
        // specifying inner exception object
        Public NegativeNumberException(stringmessage, Exception inner )  base( message, inner )
        {

        }
}
```

Programmer-Defined Exception > Example

```
try
  {
     mark = Convert.ToDouble(txtTotalMark.Text);
     lg = crs.getLetterGrade(mark);
     //calculate grade point of a course
     credit = Convert.ToInt32(txtCredit.Text);
     lgp = crs.getletterGradePoint();
     gp = crs.getGradePoint(credit, lgp);
     //display the result
     txtLetterGrade.Text = lg.ToString();
     txtGradePoint.Text = gp.ToString();
  }
 // display MessageBox if negative number input
 catch( NegativeNumberException nex)
{
  MessageBox.Show( nex.Message);
}
  catch (Exception ex)
  {
     MessageBox.Show(ex.Message +"\n"+ex.StackTrace.ToString() +"\n"+ex.GetType() );
  }
```

Handling Overflows with Operators checked and unchecked

- Calculation that could overflow
  - ✓ Use a checked context when performing calculations that can result in overflow
    - > Programmer should define exception handlers to process the overflow
  - ✓ In .NET, primitive data types are stored in fixed-size structure
    - > Example, maximum for int is 2,147,483,647
  - ✓ Overflow causes program to produce incorrect result
  - ✓ C# provides operators checked and unchecked to specify the validity of integer arithmetic.
- Checked context
  - ✓ The CLR throws an overflowException if overflow occurs during calculation
- Unchecked context
  - ✓ The result of the overflow is truncated
- Explicit conversions between integral data types can cause overflow

Handling Overflows > Example

```
class Overflow
{
        static voidMain( string[] args )
    {
            intnumber1 = Int32.MaxValue; // 2,147,483,647
           intnumber2 = Int32.MaxValue; // 2,147,483,647
           intsum = 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 overflow exception
          catch( OverflowException overflowException )
          {
                Console.WriteLine( overflowException.ToString() );
          }
          Console.WriteLine( "\nsum after checked operation: {0}", sum );
          Console.WriteLine("\nSum integers in unchecked context:");
          sum = unchecked( number1 + number2 );
          Console.WriteLine( "sum after unchecked operation: {0}", sum );
    } // end method Main
}// end class Overflow
```

Data Validation

Validating User Input

Data Validation

- Prevent users from entering invalid data whenever possible
- Guide users through the process of entering valid data
- Allow users flexibility in how and when they enter data
- Consider validation requirements when designing your application
- Place validation code in the appropriate location, depending on the requirements of your application
  - ✓ Restricting User Input
    - > Guidelines for Validating User Input
    - > Intrinsic Validation
  - ✓ Validating Field Data
  - ✓ Validating Form Data

Intrinsic Validation

- The built-in control properties and methods that you can use to restrict and validate user input
- Common controls that provide intrinsic validation:
  - ✓ RadioButton
    - > Restricts entry to On or Off
  - ✓ CheckBox
    - > Restricts entry to Checked or Unchecked
  - ✓ CheckedListBox
    - > Provides a list of valid entries
  - ✓ ListBox
    - > Provides a list of valid entries (graphics and text)
  - ✓ DateTimePicker
    - > Restricts entry to dates or times
  - ✓ MonthCalendar
    - > Restricts entry to a range of dates
  - ✓ TextBox
    - > Set properties to restrict or modify data entry
  - ✓ MaskedTextBox
    - > Set enhanced properties for masking input and formatting output

Masked Edit Control

- #
  - ✓ Digit placeholder (entry required).
- 9
  - ✓ Digit placeholder (entry optional).
- ?
  - ✓ Alphabetical character placeholder: a-z or A-Z (entry optional).
- A
  - ✓ Alphanumeric character placeholder (entry required).
- a
  - ✓ Alphanumeric character placeholder (entry optional).
- \
  - ✓ Used in input mask to indicate that a literal follows.
  - ✓ For example, to display a space in a mask, you would type a space after the \ character.

Validating Field Data

- Using Boolean Functions
- Using the ErrorProvider Component
- Set Focus on Controls and Text
- Modify User Input
- Using Validation Events

Common Boolean Function

- IsNumeric
  - ✓ Returns a Boolean value indicating whether an expression is recognized as a number
- IsDate
  - ✓ Returns a Boolean value indicating whether an expression evaluates to a valid date

```
if IsNumeric(txtCredit.Text) {
    MessageBox.Show("The text box contains a number.");
}
else {
    MessageBox.Show(“Credit should be a numeric value.");
}
```

Using ErrorProvider Component

- Add the ErrorProvider component to a form
  - ✓ Available on the Windows Forms tab of the Toolbox
- Call the SetError method
  - ✓ The first parameter specifies where the icon should appear, and
  - ✓ the second parameter specifies error message to display:

```
ErrorProvider1.SetError (Textbox1, "Please enter a valid date.”
```

  - ✓ If the user enters invalid data, an error icon and message appear on the form

Set Focus on Controls and Text

- Why set focus?
  - ✓ When a control has focus, the user can enter data for that control by using a mouse or keyboard
  - ✓ When a user enters invalid data, you can keep the focus on the appropriate control until the error is fixed
  - ✓ To set focus on a TextBox control, use Focus method:
    - > TextBox1.Focus()
  - ✓ To select all text within the control, use SelectAll:
    - > TextBox1.SelectAll( )

Using Validation Events

- Use the CausesValidation property to trigger the Validating event
- Use the CausesValidation property to trigger the Validating event for a control.
- The CausesValidation property is a Boolean property that is available on the Windows Forms controls provided by default in the Toolbox.
- Validating event
  - ✓ The Validating event is raised before a control loses focus, after the Leave event and before the Validated event.
  - ✓ The Validating event handler can contain any validation code that you need for the specific control.
  - ✓ The Validated event is triggered after the conditions in the Validating event have been met and before the LostFocus event occurs.

Validating Form Data

- Provide visual cues to the user
  - ✓ Example
    - > Disable the OK/Save button until the user enters data in all required fields
- Validate all the fields on the form at one time
  - ✓ Example
    - > Put all the validation code in the Click event handler of the OK button

Security Issues

- Authenticating users
  - ✓ Verifying the current Windows user
  - ✓ Use the UserName property of the
  - ✓ SystemInformation object
  - ✓ Example
    - MessageBox.Show("Current user is " & SystemInformation.UserName)
- Securing your code

For more information

- Deitel, C#-How to Program. USA, 2010
- Svetlin Nakov *et al*. Fundamentals of Computer Programming With C#. Sofia, 2013
- Joel Murach, Anne Boehm. Murach C# 2012, Mike Murach & Associates Inc USA, 2013

QUESTION?