Topic 3C

Data Validation and Exceptions

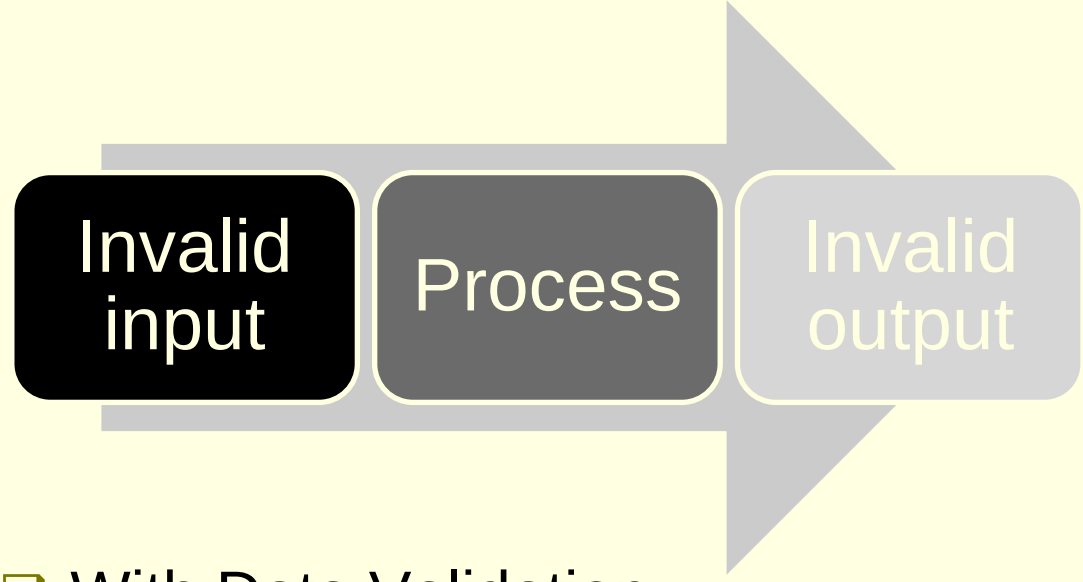
Topics

Objectives:

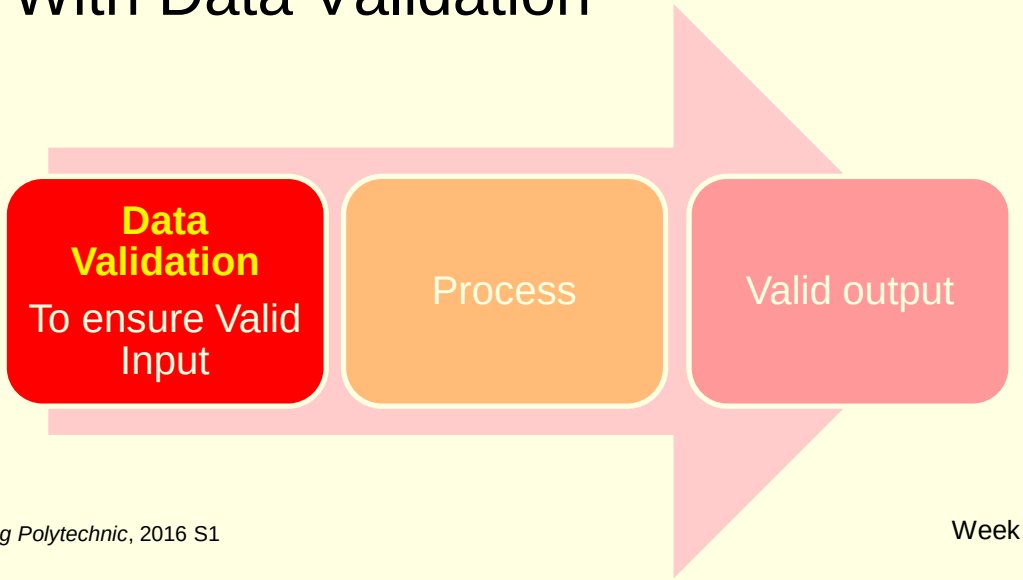
- Understand how to validate data input by user
- Understand how to handle error during runtime
- Apply GUI design with validation
- Use TryParse method
- Apply range checking
- Handle exceptions in code

Validating Data

- Anticipate user will enter invalid data
- Write code to prevent invalid data from being used in the program
- It results in invalid output

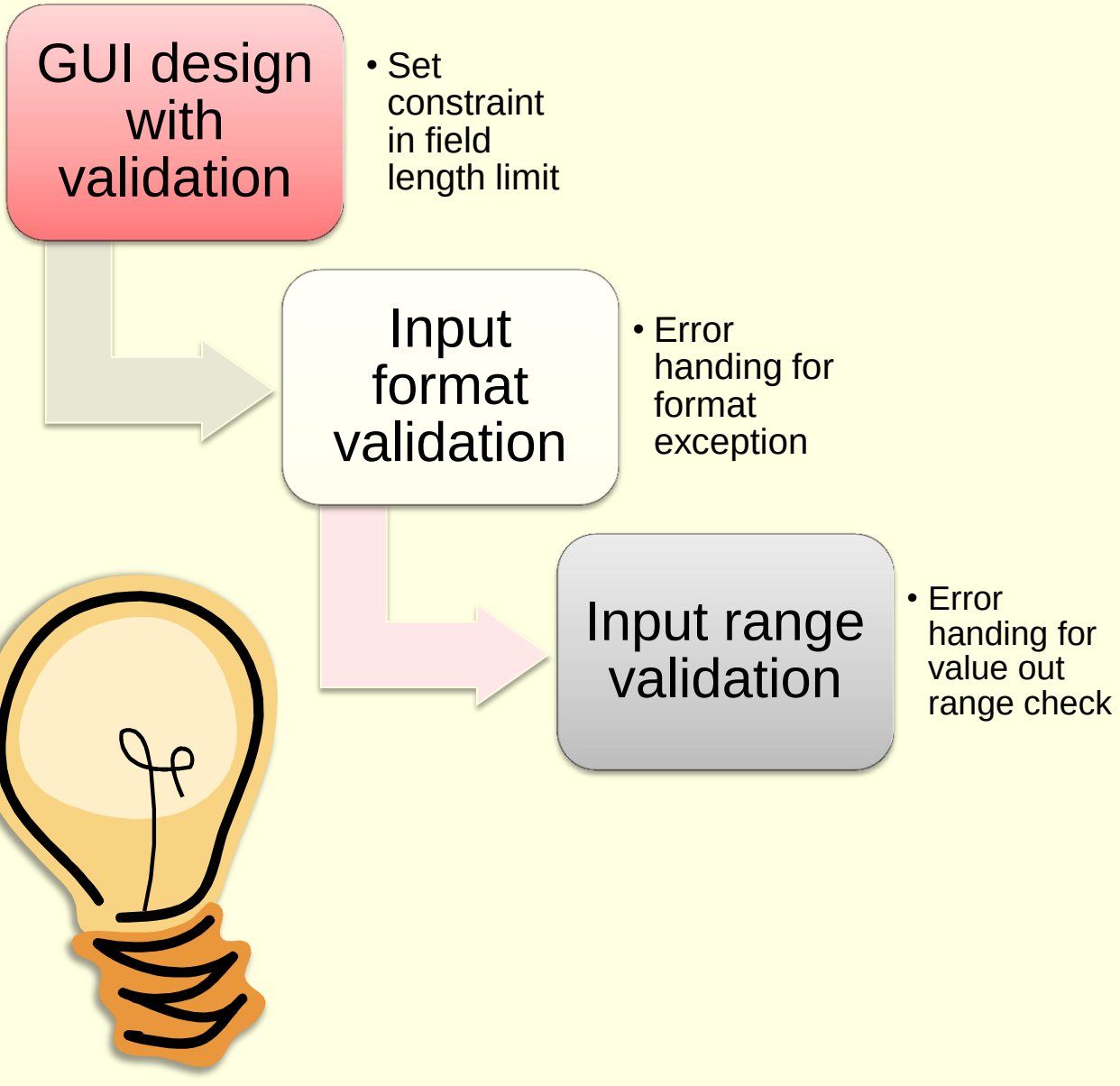


- With Data Validation

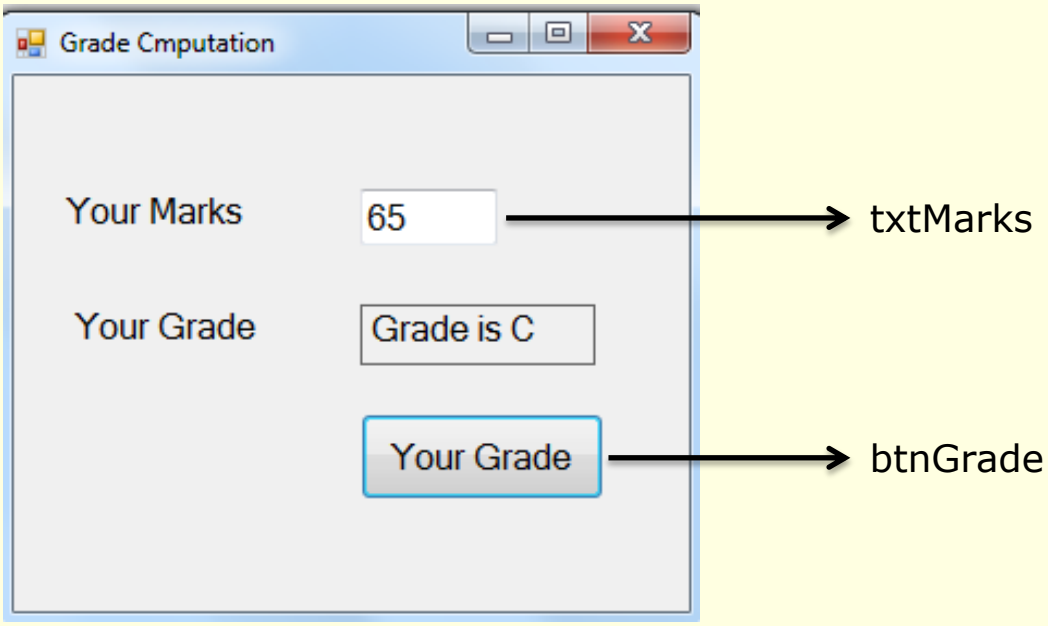


Validating Data

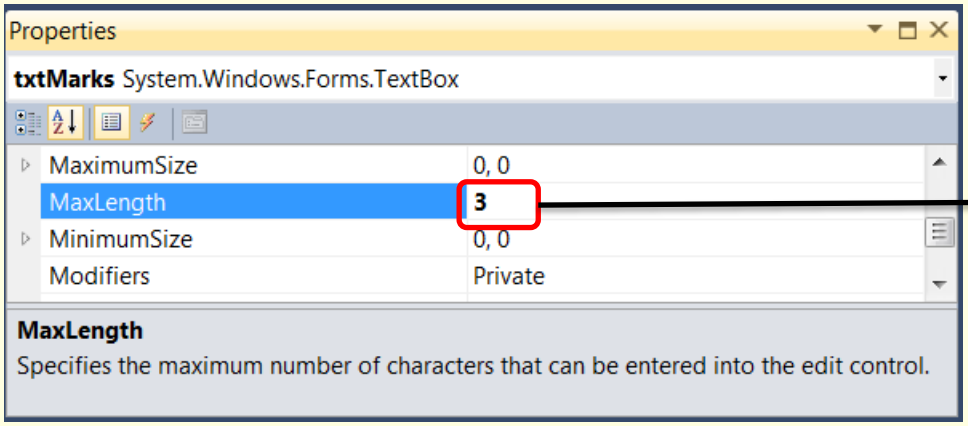
- 3 levels of data validation



Example 1: Grade Computation with Data validation

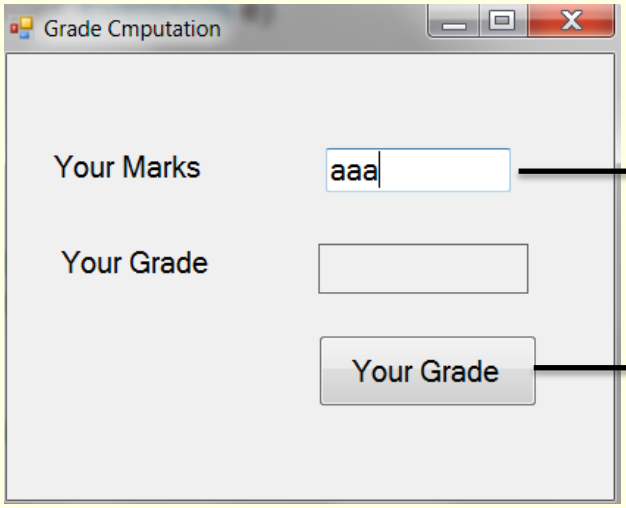


- Validate mark entered by user
  - **GUI design with validation**
    - **Limit the Length of Mark text box through Properties Setting window**
    - **Set it to 3. User will NOT be able to enter more than 3 characters for Mark field (e.g 100)**



Example 1: Grade Computation with Data validation

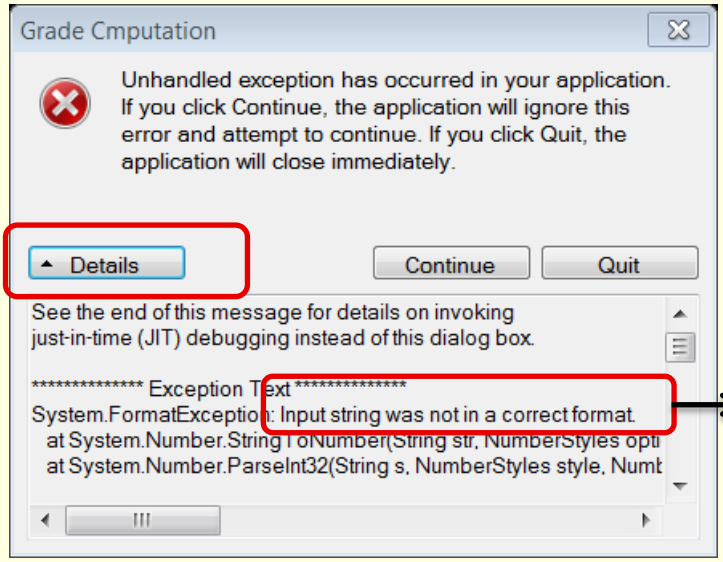
- However, when user runs the program user may enter invalid data with 3 characters
- E.g



User enter aaa for mark

When user clicks on button

- It throws a format exception during runtime.



Click on Details to see the exception error message

String format is incorrect. It is expecting numeric but "aaa" is NOT a correct numeric format

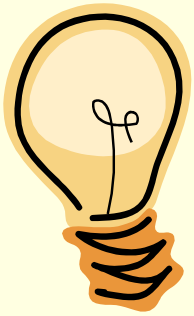
Example 1: Grade Computation with Data validation

- Which method throws the format exception?

```
private void btnGrade_Click(object sender, EventArgs e)
{
    int marks=0;

    marks = int.Parse(txtMarks.Text);
```

- The parsing method:
  - **int.Parse()**
  - The conversion process cannot take place because the argument (e.g txtMark.Text) passed is not a numeric value (e.g 123)
  - Hence, it throws a format exception

- **Validate Input format**



  - To handle the format exception
  - Use **int.TryParse()**
  - TryParse will convert the string representation into numeric and check if the conversion is successful

Example 1: Grade Computation with Data validation

- int.**TryParse** () checks the input format and stores converted value after parsing
- **TryParse** () is available for data type as such float.TryParse, double.TryParse
- int.**TryParse** will replace int.Parse()
  - Important notes in *** - see below,** more in the next slide
- Validate input format of Mark:
  - User enters **Mark** as **non numeric or empty**
  - Program will inform user of the invalid input
  - Allow user to re-enter a proper value
  - when user clicks on **Your Grade** button, the program will valid the input format.
  - Validation code is in **btnGrade_Click** method

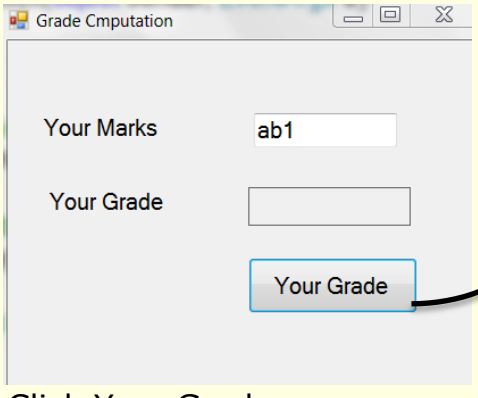
```
private void btnGrade_Click(object sender, EventArgs e)
{
   int marks=0;

   // Validate if the value is numeric and store numeric result in marks
1  if (int.TryParse(txtMarks.Text, out marks) == false)
   {
      // display message box
2     MessageBox.Show("Please enter a value in numeric");

      // clear text box and set cursor focus
3     txtMarks.Text = "";
      txtMarks.Focus();
   }
   // valid numeric value is stored in marks
```

* MUST set marks = 0

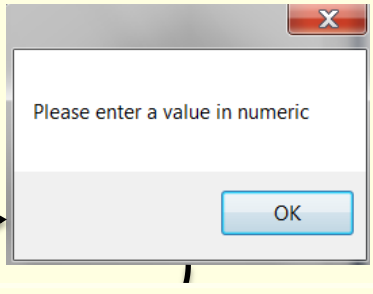
MUST add in keyword **out before marks .** TryParse needs it to store the value

Example 1: Grade Computation with Data validation

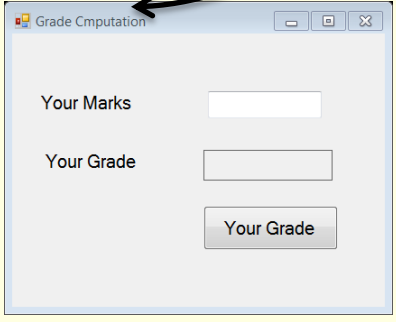
- Explain code in:
  1. Bool int.TryParse(string s, **out** int result)
     - It will return a false if the value in s (e.g txtMarks.Text) contains non numeric data
     - When it failed, the program will execute 2 3
     - If it is successful, it stores the converted value into variable marks
     - **out** marks, **out** is the keyword used for variable to store the value. To use out variable, mark needs to be set as 0.
  2. It pops up message box with error message
  3. It clears the txtMarks field and place the cursor at txtMarks field. It allows user to re-enter the mark's value.



Click Your Grade.
Message box shows error

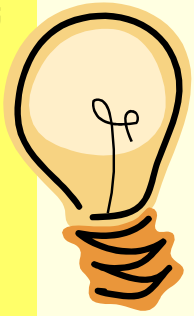


Click Ok. Mark field is cleared for user to re-enter value



Example 1: Grade Computation with Data validation

Validate Input Range



- Check user enters value within range
- Validate input range of Mark (0-100):
  - User enters **Mark** out of range e.g -1, 101
  - Program will inform user of the invalid input
  - Allow user to re-enter a proper value
  - when user clicks on **Your Grade** button, the program will valid the input.
  - Validation code is in **btnGrade_Click** method

1
2
3

```
// valid numeric value is stored in marks
    |
 //Validate range of mark
else if (marks < 0 || marks > 100)
{
   // display message box
   MessageBox.Show("Please enter a value from 0-100");

   // clear text box and set cursor focus
   txtMarks.Text = "";
   txtMarks.Focus();
}
```

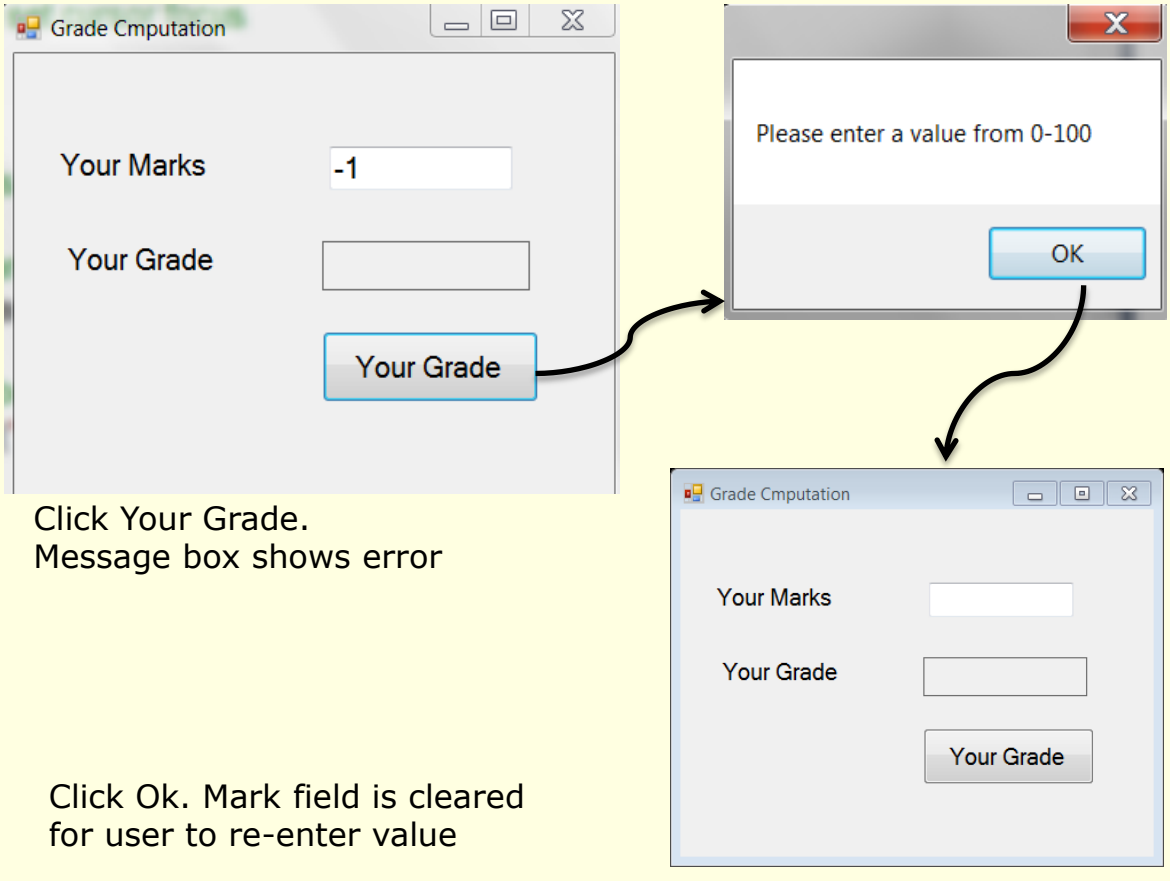
Example 1: Grade Computation with Data validation

1 ❑ After the input format is validated successfully
- ❑ Input value is stored in variable marks

❑ Using if statement to check the value

2
- ❑ If the value is out of range (e.g -1 or 101)
- ❑ It pops up message box with error message

3
- ❑ It clears the txtMarks field and place the cursor at txtMarks field. It allows user to re-enter the mark's value.



Click Your Grade.
Message box shows error

Click Ok. Mark field is cleared for user to re-enter value

Example 1: Grade Computation with Data validation

Complete source with data validation

```
private void btnGrade_Click(object sender, EventArgs e)
      {
         int marks=0;

         // Validate if the value is numeric and store numeric result in marks
         if (int.TryParse(txtMarks.Text, out marks) == false)
         {
            // display message box
            MessageBox.Show("Please enter a value in numeric");

            // clear text box and set cursor focus
            txtMarks.Text = "";
            txtMarks.Focus();
         }
         // valid numeric value is stored in marks

          //Validate range of mark
         else if (marks < 0 || marks > 100)
         {
            // display message box
            MessageBox.Show("Please enter a value from 0-100");

            // clear text box and set cursor focus
            txtMarks.Text = "";
            txtMarks.Focus();
         }

         //  Valid input with correct format and range
         // Compute Grade
         else
         {
            if (marks >= 80)
               lblResult.Text = "Grade is A";
            else if (marks >= 70)
               lblResult.Text = "Grade is B";
            else if (marks >= 60)
               lblResult.Text = "Grade is C";
            else if (marks >= 50)
               lblResult.Text = "Grade is D";
            else
               lblResult.Text = "Grade is F";
         }
      }
```

Compute grade when input is valid

Runtime Errors

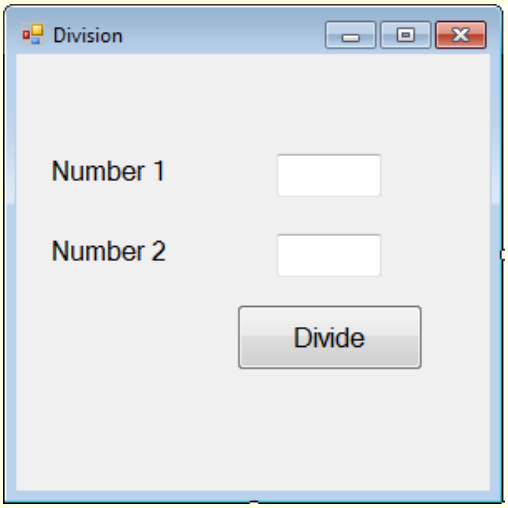
- When program processes numeric data entered by a user, several errors can happen during runtime.

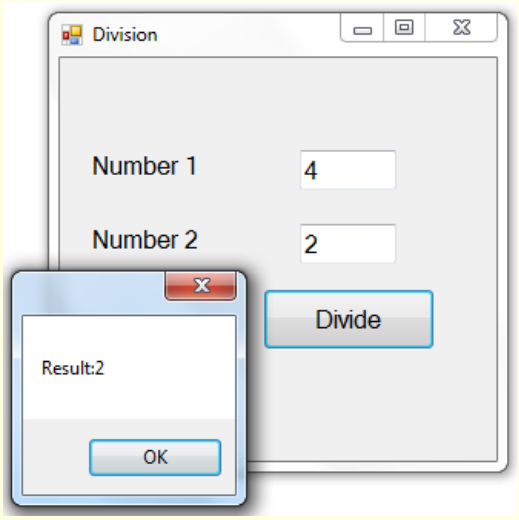
- The three errors that occur most often are:
  - **Format Exception**
    - Occurs when user enters data that a statement within the program cannot process properly
    - **TryParse will help to capture this error**
  - **Overflow Exception**
    - Occurs when user enters a value greater than the maximum value that can be process by the statement
  - **Divide By Zero Exception**
    - It is not possible to divide by zero
    - Occurs when your program contains a division operation and the divisor is equal to zero

Exception Handling

- Exceptions are unexpected conditions in a program
- Exception examples:
  - Opening file that does not exist
  - Dividing a number by Zero
- Exceptions are handled in a try-catch block.
- Let's look at a simple example without exception handling.

Exception Handling



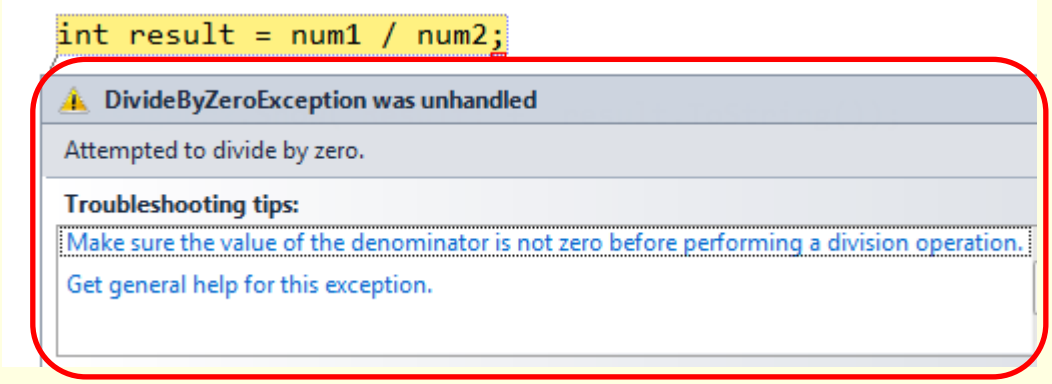



```
private void btnDivide_Click(object sender, EventArg
{
    int num1 = int.Parse(txtNumber1.Text);
    int num2 = int.Parse(txtNumber2.Text);

    int result = num1 / num2;

    MessageBox.Show("Result:"+  result.ToString());
}
```

- The above example shows when 4 is divided by 2, the result of 2 is displayed.
- What happens when 4 is divided by 0?

Exception Handling

- An exception is thrown by .NET as shown:



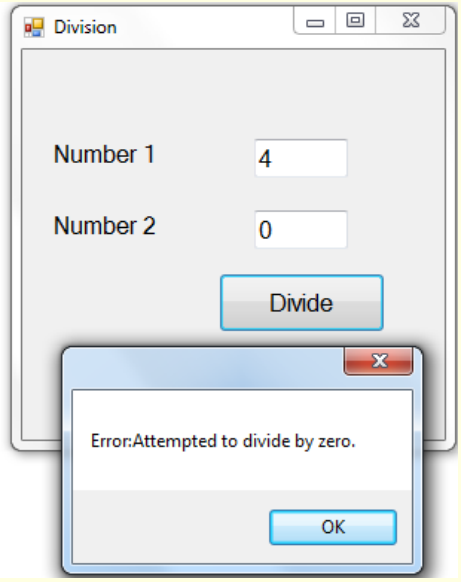
- To 'catch' exceptions such as these, we use the following:

```
try
  {
      //code that throws exception e
  }
  catch (Exception e)
  {
      //code that handles exception e
  }
```

Exception Handling

- Here, we 'catch' the error and display a message to the user

```
int num1 = int.Parse(txtNumber1.Text);
int num2 = int.Parse(txtNumber2.Text);
try
{
    //code that throws exception e
    int result = num1 / num2;
    MessageBox.Show("Result:" + result.ToString());
}
catch (Exception ex)
{
    //code that handles exception e
    // Display error
    MessageBox.Show("Error:" + ex.Message);
}
```



Catching Multiple Exceptions

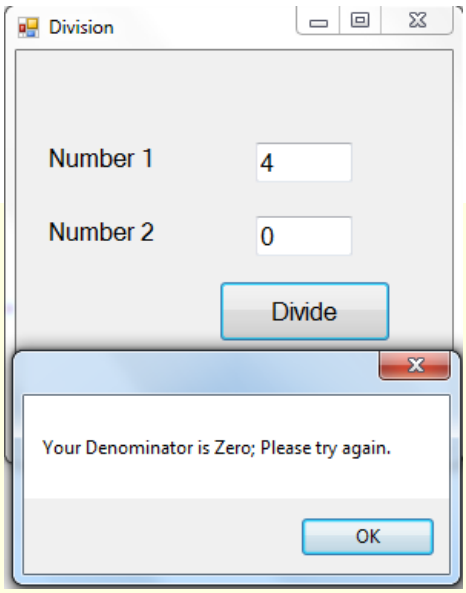
- It is possible to catch multiple exceptions in a catch block
- Order of exceptions is important as more generic exceptions should be handled at the end

```
try
  {
      //code that throws exception e1
      //code that throws exception e2
  }
  catch(MyException e1)
  {
      //code that handles exception e1
  }
  catch(Exception e2)
  {
      //code that handles exception e2
  }
```

Catching Multiple Exceptions

- In this example, the more specific error DivideByZeroException is caught first:

```
int num1 = int.Parse(txtNumber1.Text);
int num2 = int.Parse(txtNumber2.Text);
try
{
    //code that throws exception e
    int result = num1 / num2;
    MessageBox.Show("Result:" + result.ToString());
}
catch (DivideByZeroException ex)
{
    // handles more specific error
     MessageBox.Show("Your Denominator is Zero; Please try again." );
     txtNumber2.Clear();
     txtNumber2.Focus();
}
catch (Exception ex)
{
    //handles a more general error
    MessageBox.Show("Error:" + ex.Message);
}
```



finally block

- Executes always at the end after the last catch block
  - Commonly used for cleaning up resources, examples : connections to database and files etc.

```
try
 {
   //code that throws exception e1
   //code that throws exception e2
 }
 catch (MyException e1)
 {
   //code that handles exception e1
 }
 catch (Exception e2)
 {
   //code that handles exception e2
 }
 finally
 {
   //clean up code, close resources
 }
```

Summary

- The if/else control enables the computer to do different actions depending on certain Test Conditions.
- The key control structure is:

```
if ( test is true )
      {
            // perform these statements
      }else if (test is true )
      {
            // perform these statements
      }else
      {
            // perform these statements only if
            // the rest are  all not true
      }
```

- We apply if/else in data validation
  - Format error using TryParse
  - Range error using value checking
  - Valid input then process with computation and other logic

- Exceptions are unexpected conditions in a program which can be handled in your code using try/catch

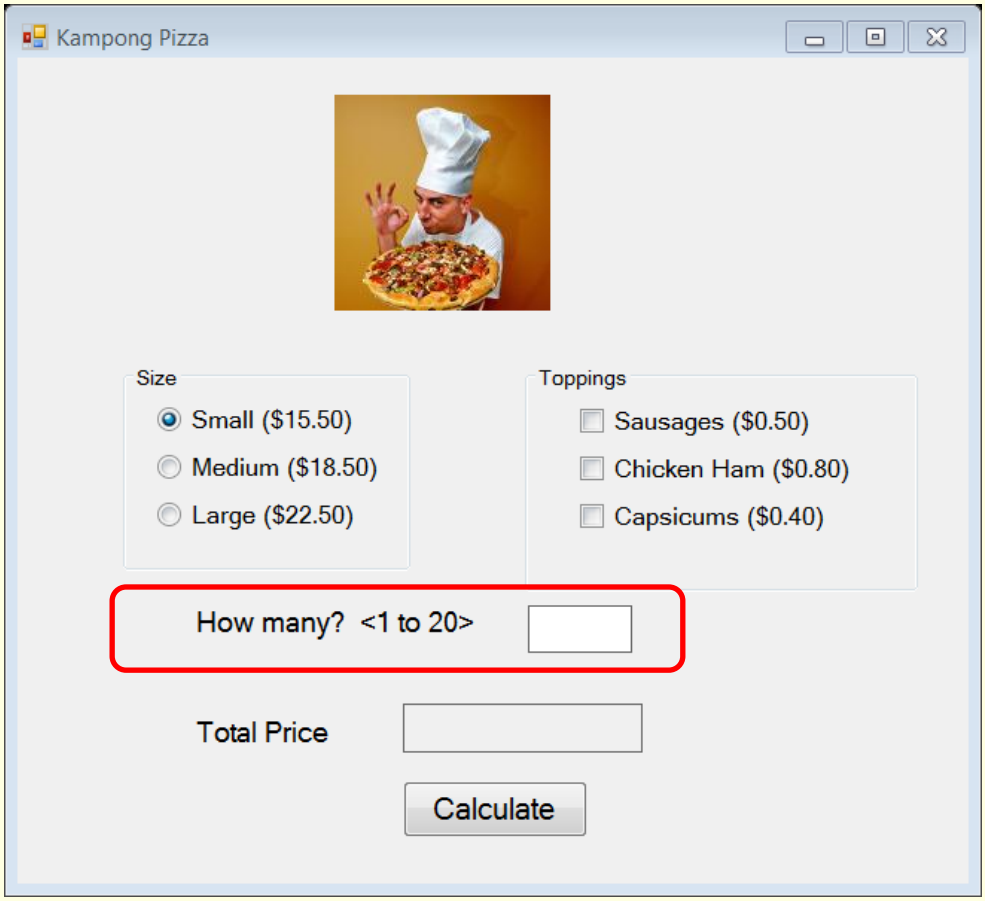
Practical 3C

- Tutor will distribute Top3Cdemo files
- It contains the demo program and startup files for Practical 3C
- These applications are similar to Practical 3B
- But in this practical, we will focus in the data validation process for user input

Practical 3C

Question 1: Pizza Ordering System with Data Validation

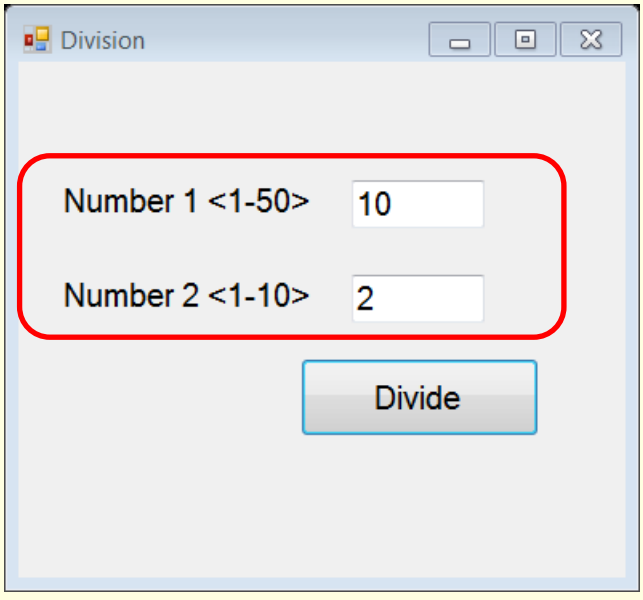
- Let us apply data validation to the Pizza Ordering System input field.
- Add in data validation:
  - Number of pizza
    - 1-20 and it must be numeric
    - Display error message and allow user to re-enter



Practical 3C

Question 2: Division Application with Data Validation

- Using Division Application
- Let us apply data validation to the Division Application for all user input.
- Add in the data validation for
  - Number 1: range from 1-50
  - Number 2: range from 1-10
  - Display error message and allow user to re-enter



Practical 3C

Question 3:Debug Your Program

- Open the example 1: Grade Computation with Data validation
- Modify the code in btnGrade_Click

```
private void btnGrade_Click(object sender, EventArgs e)
      {
            ....
        //Validate range of mark
       else if (marks < 0 || marks > 100)
       {
          // display message box
          MessageBox.Show("Please enter a value from 0-100");

          // clear text box and set cursor focus
          txtMarks.Text = "";
          txtMarks.Focus();
       }
            ...
```

Modify the condition to: else if (marks <0 && marks>100)

- Run the application. Check the result. Write your finding.

- Using debugger in VS, set a break point at the line that you have changed. Monitor the marks value and observe the result. Explain the difference between using || and &&

End of Topic 3C

Data Validation and Exceptions