Chapter 7

How to handle exceptions and validate data

Objectives

Applied

1. Given a form that uses text boxes to accept data from the user, write code that catches any exceptions that might occur.
2. Given a form that uses text boxes to accept data and the validation specifications for that data, write code that validates the user entries.
3. Use dialog boxes as needed within your applications.

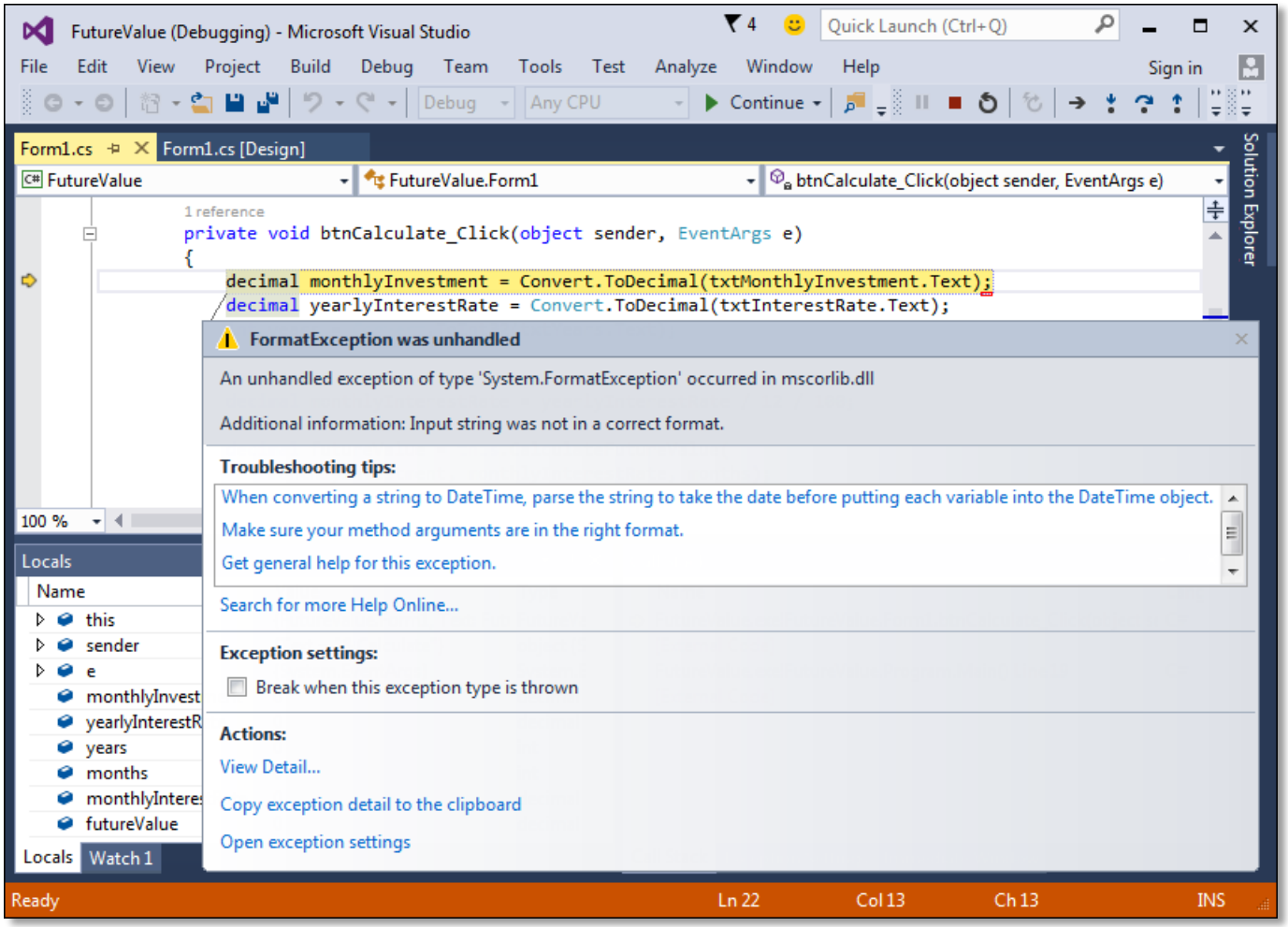
Knowledge

1. Explain what an exception is and what it means for an exception to be thrown and handled.
2. Describe the Exception hierarchy and name two of its subclasses.
3. Describe the use of try-catch statements to catch specific exceptions as well as all exceptions.

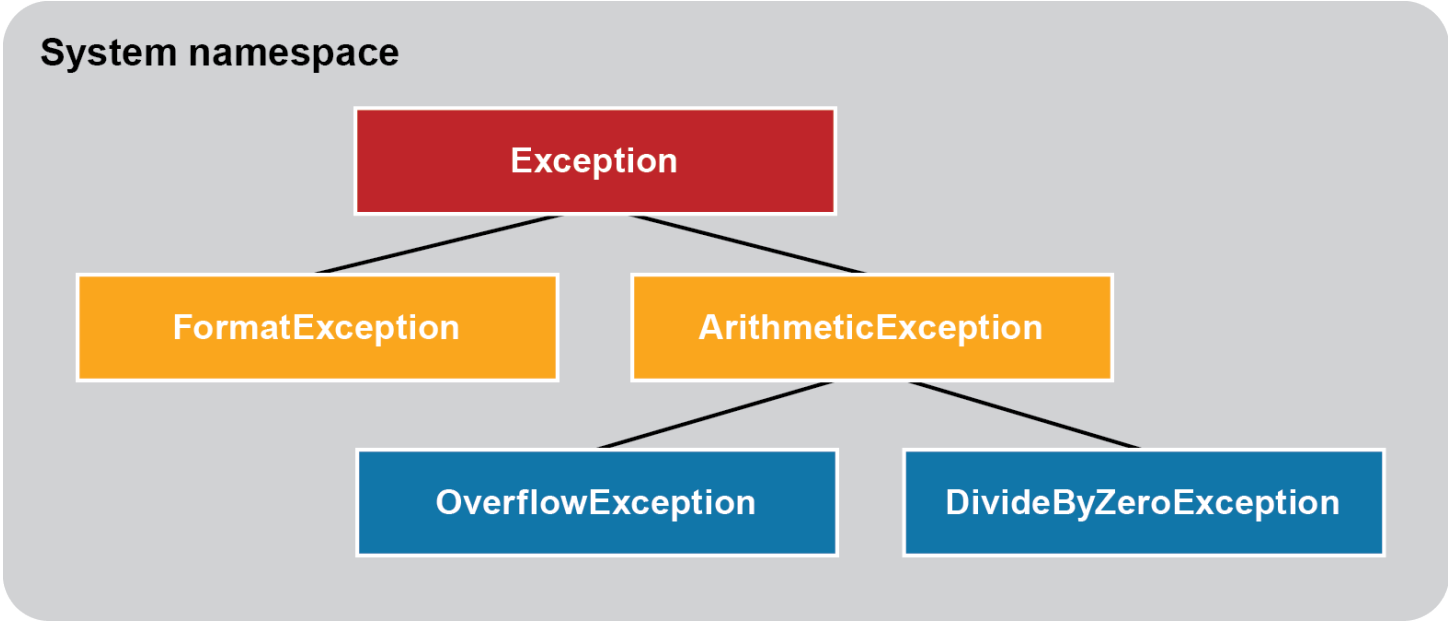
Objectives (cont.)

4. Describe the use of the properties and methods of an exception object.
5. Describe the use of throw statements.
6. Describe the three types of data validation that you're most likely to perform on a user entry.
7. Describe two ways that you can use generic validation methods in a method that validates all of the user entries for a form.

The dialog box for an unhandled exception



The Exception hierarchy for five common exceptions



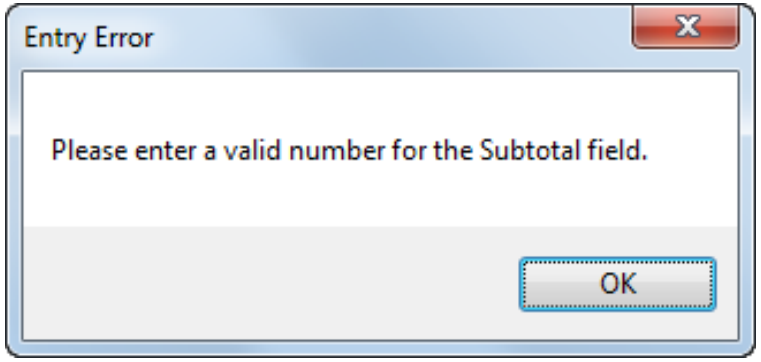
Methods that might throw exceptions

Class	Method	Exception
`Convert`	`ToDecimal(string)`	`FormatException` `OverflowException`
`Convert`	`ToInt32(string)`	`FormatException` `OverflowException`
`Decimal`	`Parse(string)`	`FormatException` `OverflowException`
`DateTime`	`Parse(string)`	`FormatException`

The syntax to display a dialog box with an OK button

```
MessageBox.Show(text[, caption]);
```

A dialog box with an OK button



The statement that displays this dialog box

```
MessageBox.Show(
    "Please enter a valid number for the Subtotal field.",
    "Entry Error");
```

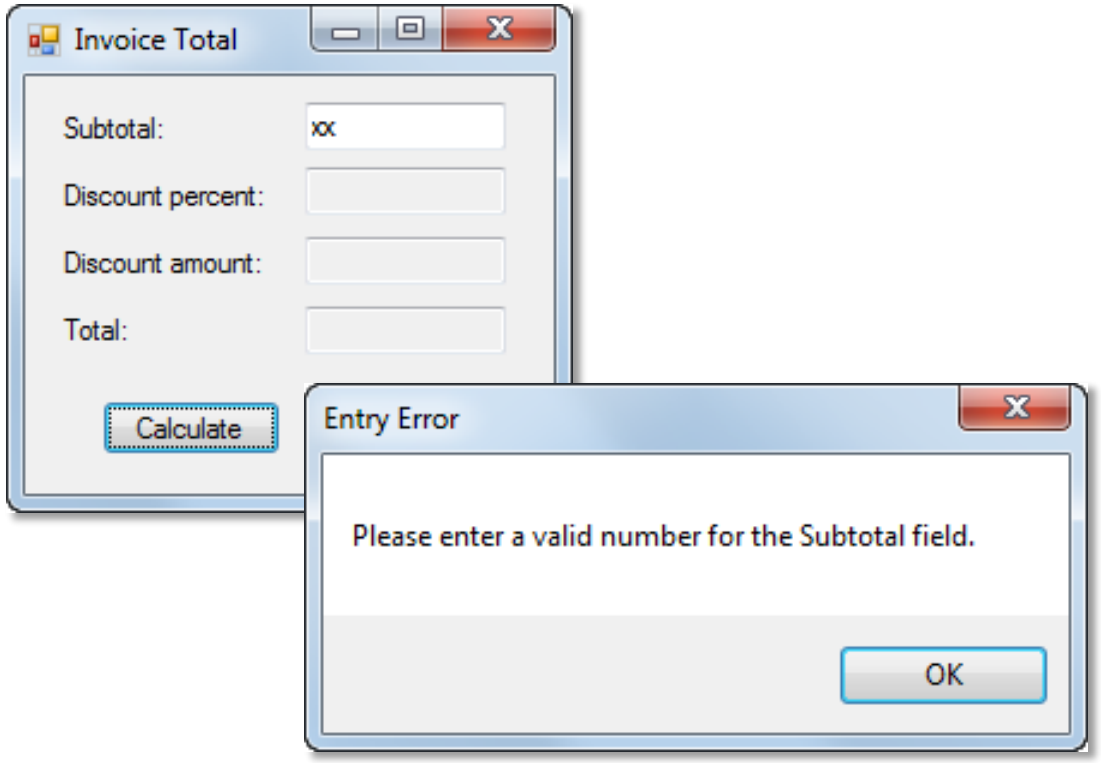
The syntax for a simple try-catch statement

```
try { statements }
catch { statements }
```

A try-catch statement

```
try
{
    decimal subtotal = Convert.ToDecimal(txtSubtotal.Text);
    decimal discountPercent = .2m;
    decimal discountAmount = subtotal * discountPercent;
    decimal invoiceTotal = subtotal - discountAmount;
}
catch
{
    MessageBox.Show(
        "Please enter a valid number for the Subtotal field.",
        "Entry Error");
}
```

The dialog box that's displayed for an exception



The syntax for a try-catch statement that accesses the exception

```
try { statements }
catch(ExceptionClass exceptionName) { statements }
```

Two common properties for all exceptions

Property	Description
`Message`	Gets a message that briefly describes the current exception.
`StackTrace`	Gets a string that lists the methods that were called before the exception occurred.

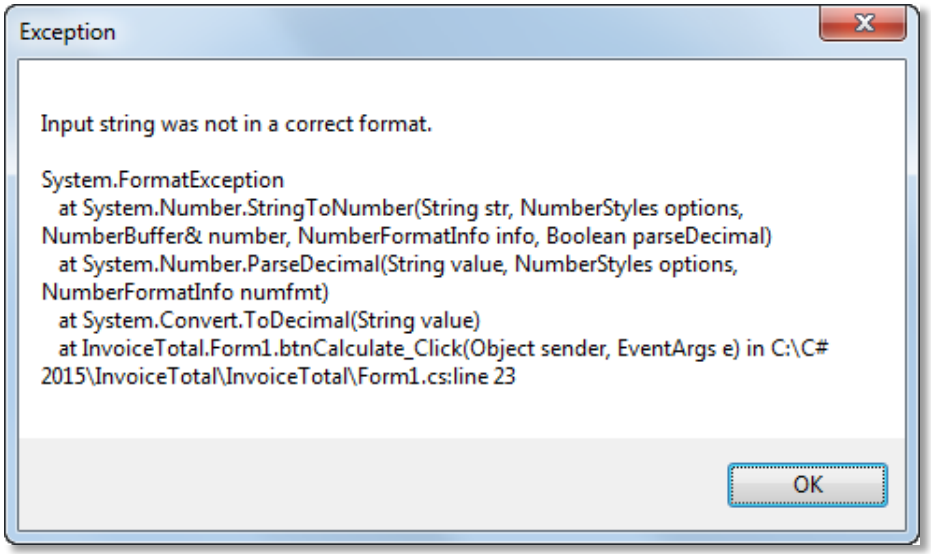
A common method for all exceptions

Method	Description
`GetType()`	Gets the type of the current exception.

A try-catch statement that accesses an exception

```
try
{
    decimal subtotal =
        Convert.ToDecimal(txtSubtotal.Text);
}
catch(Exception ex)
{
    MessageBox.Show(ex.Message + "\n\n" +
        ex.GetType().ToString() + "\n" +
        ex.StackTrace, "Exception");
}
```

The dialog box that's displayed for an exception



The syntax for a try-catch statement that catches specific types of exceptions

```
try { statements }
[catch(MostSpecificException [exceptionName]) { statements }]...
[catch(NextMostSpecificException [exceptionName]) { statements }]...
[catch([LeastSpecificException [exceptionName]]) { statements }]
[finally { statements }]
```

A statement that catches two specific exceptions

```
try
{
    decimal monthlyInvestment =
        Convert.ToDecimal(txtMonthlyInvestment.Text);
    decimal yearlyInterestRate =
        Convert.ToDecimal(txtInterestRate.Text);
    int years = Convert.ToInt32(txtYears.Text);
}
catch(FormatException)   // a specific exception
{
    MessageBox.Show(
        "A format exception has occurred. Please check all entries.",
        "Entry Error");
}
catch(OverflowException) // another specific exception
{
    MessageBox.Show(
        "An overflow exception has occurred. Please enter smaller values.",
        "Entry Error");
}
catch(Exception ex)     // all other exceptions
{
    MessageBox.Show(ex.Message, ex.GetType().ToString());
}
finally      // this code runs whether or not an exception occurs
{
    PerformCleanup();
}
```

The syntax for throwing a new exception

```
throw new ExceptionClass([message]);
```

The syntax for throwing an existing exception

```
throw exceptionName;
```

When to throw an exception

- When a method encounters a situation where it isn't able to complete its task.
- When you want to generate an exception to test an exception handler.
- When you want to catch the exception, perform some processing, and then throw the exception again.

A method that throws an exception when an exceptional condition occurs

```
private decimal CalculateFutureValue(
    decimal monthlyInvestment,
    decimal monthlyInterestRate, int months)
{
    if (monthlyInvestment <= 0)
       throw new Exception(
           "Monthly Investment must be greater than 0.");
    if (monthlyInterestRate <= 0)
       throw new Exception(
           "Interest Rate must be greater than 0.");
        .
        .
}
```

Code that throws an exception for testing

```
try
{
    decimal subtotal = Convert.ToDecimal(txtSubtotal.Text);
    throw new Exception("An unknown exception occurred.");
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message + "\n\n"
        + ex.GetType().ToString() + "\n"
        + ex.StackTrace, "Exception");
}
```

Code that rethrows an exception

```
try
{
    Convert.ToDecimal(txtSubtotal.Text);
}
catch (FormatException fe)
{
    txtSubtotal.Focus();
    throw fe;
}
```

The code for the Future Value application with exception handling

```
private void btnCalculate_Click(object sender, EventArgs e)
{
    try
    {
        decimal monthlyInvestment =
            Convert.ToDecimal(txtMonthlyInvestment.Text);
        decimal yearlyInterestRate =
            Convert.ToDecimal(txtInterestRate.Text);
        int years = Convert.ToInt32(txtYears.Text);

        decimal monthlyInterestRate = yearlyInterestRate / 12 / 100;
        int months = years * 12;

        decimal futureValue = this.CalculateFutureValue(
            monthlyInvestment, monthlyInterestRate, months);
        txtFutureValue.Text = futureValue.ToString("c");
        txtMonthlyInvestment.Focus();
    }
```

The code for the Future Value application with exception handling (cont.)

```
    catch(FormatException)
    {
        MessageBox.Show(
            "Invalid numeric format. Please check all entries.",
            "Entry Error");
    }
    catch(OverflowException)
    {
        MessageBox.Show(
            "Overflow error. Please enter smaller values.",
            "Entry Error");
    }
    catch(Exception ex)
    {
        MessageBox.Show(
            ex.Message,
            ex.GetType().ToString());
    }
}
```

The code for the Future Value application with exception handling (cont.)

```
private decimal CalculateFutureValue(decimal monthlyInvestment,
    decimal monthlyInterestRate, int months)
{
    decimal futureValue = 0m;
    for (int i = 0; i < months; i++)
    {
        futureValue = (futureValue + monthlyInvestment)
            * (1 + monthlyInterestRate);
    }

    return futureValue;
}
```

Code that checks that an entry has been made

```
if (txtMonthlyInvestment.Text == "")
{
    MessageBox.Show(
        "Monthly Investment is a required field.",
        "Entry Error");
    txtMonthlyInvestment.Focus();
}
```

Code that checks an entry for a valid decimal value

```
decimal monthlyInvestment = 0m;
if (!(Decimal.TryParse(txtMonthlyInvestment.Text,
    out monthlyInvestment)))
{
    MessageBox.Show(
        "Monthly Investment must be a numeric value.",
        "Entry Error");
    txtMonthlyInvestment.Focus();
}
```

Code that checks an entry for a valid range

```
decimal monthlyInvestment =
    Convert.ToDecimal(txtMonthlyInvestment.Text);
if (monthlyInvestment <= 0)
{
    MessageBox.Show(
        "Monthly Investment must be greater than 0.",
        "Entry Error");
    txtMonthlyInvestment.Focus();
}
else if (monthlyInvestment >= 1000)
{
    MessageBox.Show(
        "Monthly Investment must be less than 1,000.",
        "Entry Error");
    txtMonthlyInvestment.Focus();
}
```

A method that checks for a required field

```
public bool IsPresent(TextBox textBox, string name)
{
    if (textBox.Text == "")
    {
        MessageBox.Show(name + " is a required field.",
            "Entry Error");
        textBox.Focus();
        return false;
    }
    return true;
}
```

A method that checks for a valid numeric format

```
public bool IsDecimal(TextBox textBox, string name)
{
    decimal number = 0m;
    if (Decimal.TryParse(textBox.Text, out number))
    {
        return true;
    }
    else
    {
        MessageBox.Show(
            name + " must be a decimal value.",
            "Entry Error");
        textBox.Focus();
        return false;
    }
}
```

A method that checks for a valid numeric range

```
public bool IsWithinRange(TextBox textBox, string name,
    decimal min, decimal max)
{
    decimal number = Convert.ToDecimal(textBox.Text);
    if (number < min || number > max)
    {
        MessageBox.Show(name + " must be between " +
            min.ToString()+ " and " + max.ToString() +
            ".", "Entry Error");
        textBox.Focus();
        return false;
    }
    return true;
}
```

Code that checks the validity of one entry

```
if (IsPresent(txtMonthlyInvestment,
        "Monthly Investment") &&
    IsDecimal(txtMonthlyInvestment,
        "Monthly Investment") &&
    IsWithinRange(txtMonthlyInvestment,
        "Monthly Investment", 1, 1000))
{
    MessageBox.Show(
        "Monthly Investment is valid.", "Test");
}
```

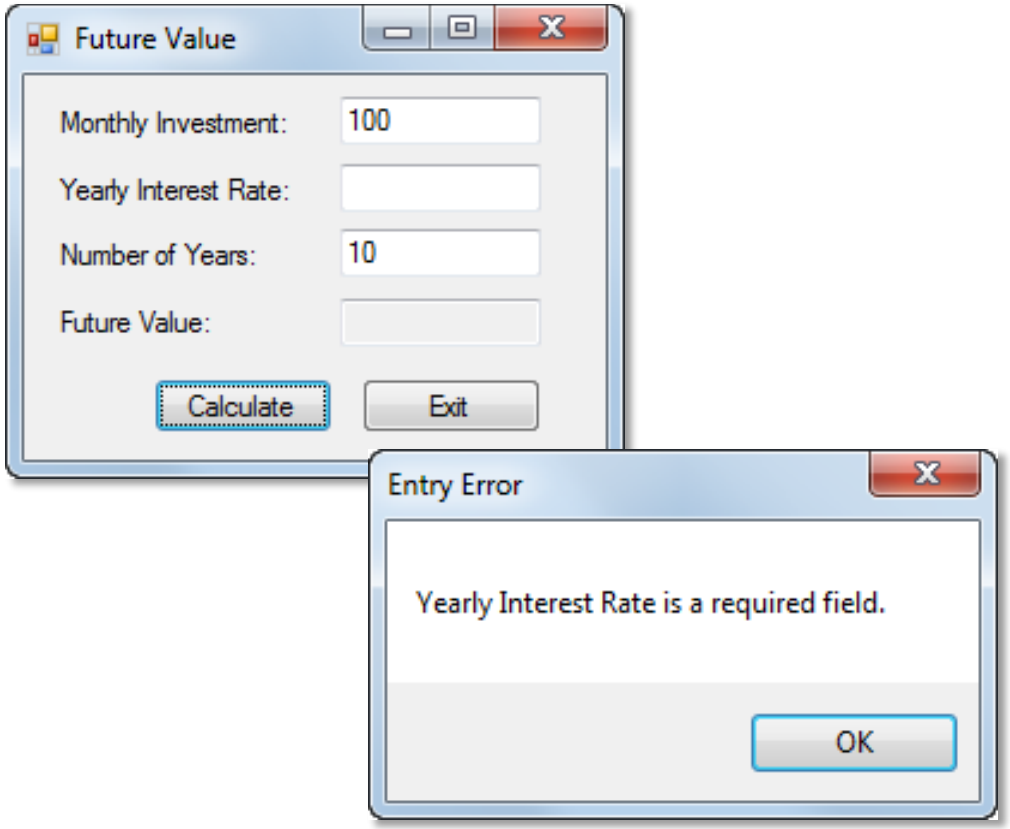
Code that uses a series of simple if statements

```
public bool IsValidData()
{
    // Validate the Monthly Investment text box
    if (!IsPresent(txtMonthlyInvestment, "Monthly Investment"))
        return false;
    if (!IsDecimal(txtMonthlyInvestment, "Monthly Investment"))
        return false;
    if (!IsWithinRange(txtMonthlyInvestment,
            "Monthly Investment", 1, 1000))
        return false;

    // Validate the Interest Rate text box
    if (!IsPresent(txtInterestRate, "Interest Rate"))
        return false;
    if (!IsDecimal(txtInterestRate, "Interest Rate"))
        return false;
    if (!IsWithinRange(txtInterestRate, "Interest Rate", 1, 20))
        return false;

    return true;
}
```

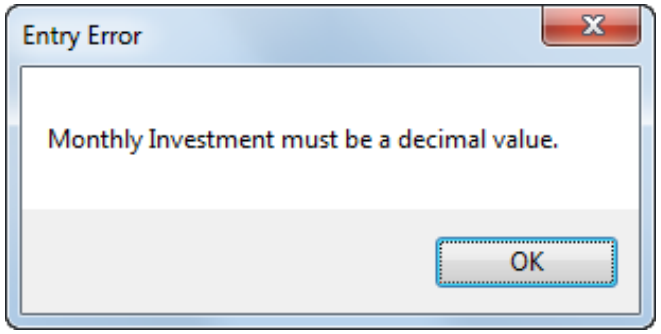
Compound conditions in a single return statement

```
public bool IsValidData()
{
    return

        // Validate the Monthly Investment text box
        IsPresent(txtMonthlyInvestment, "Monthly Investment") &&
        IsDecimal(txtMonthlyInvestment, "Monthly Investment") &&
        IsWithinRange(txtMonthlyInvestment, "Monthly Investment",
            1, 1000) &&

        // Validate the Interest Rate text box
        IsPresent(txtInterestRate, "Yearly Interest Rate") &&
        IsDecimal(txtInterestRate, "Yearly Interest Rate") &&
        IsWithinRange(txtInterestRate, "Yearly Interest Rate",
            1, 20);

}
```

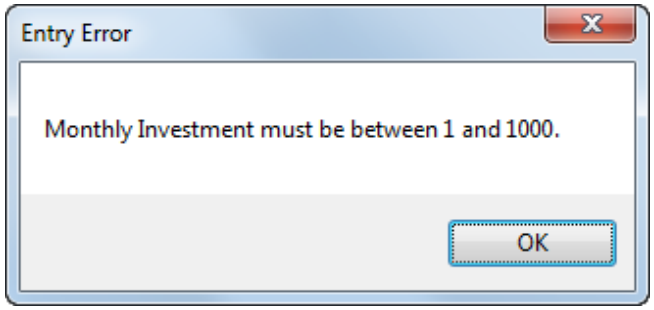
The Future Value form with a dialog box for required fields



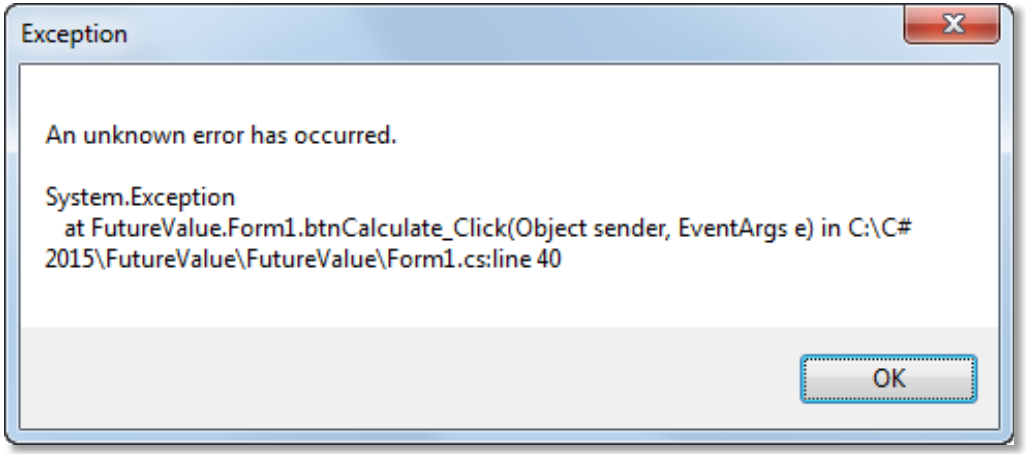
The dialog box for invalid decimals



The dialog box for invalid ranges



The dialog box for an unanticipated exception



The code for the Future Value application

```
private void btnCalculate_Click(object sender, EventArgs e)
{
    try
    {
        if (IsValidData())
        {
            decimal monthlyInvestment =
                Convert.ToDecimal(txtMonthlyInvestment.Text);
            decimal yearlyInterestRate =
                Convert.ToDecimal(txtInterestRate.Text);
            int years = Convert.ToInt32(txtYears.Text);

            int months = years * 12;
            decimal monthlyInterestRate = yearlyInterestRate / 12 / 100;
            decimal futureValue = CalculateFutureValue(
                monthlyInvestment, monthlyInterestRate, months);

            txtFutureValue.Text = futureValue.ToString("c");
            txtMonthlyInvestment.Focus();
        }
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message + "\n\n" +
            ex.GetType().ToString() + "\n" +
            ex.StackTrace, "Exception");
    }
}
```

The code for the Future Value application (cont.)

```
public bool IsValidData()
{
    return
        IsPresent(txtMonthlyInvestment, "Monthly Investment") &&
        IsDecimal(txtMonthlyInvestment, "Monthly Investment") &&
        IsWithinRange(txtMonthlyInvestment, "Monthly Investment", 1, 1000) &&

        IsPresent(txtInterestRate, "Yearly Interest Rate") &&
        IsDecimal(txtInterestRate, "Yearly Interest Rate") &&
        IsWithinRange(txtInterestRate, "Yearly Interest Rate", 1, 20) &&

        IsPresent(txtYears, "Number of Years") &&
        IsInt32(txtYears, "Number of Years") &&
        IsWithinRange(txtYears, "Number of Years", 1, 40);
}

public bool IsPresent(TextBox textBox, string name)
{
    if (textBox.Text == "")
    {
        MessageBox.Show(name + " is a required field.", "Entry Error");
        textBox.Focus();
        return false;
    }
    return true;
}
```

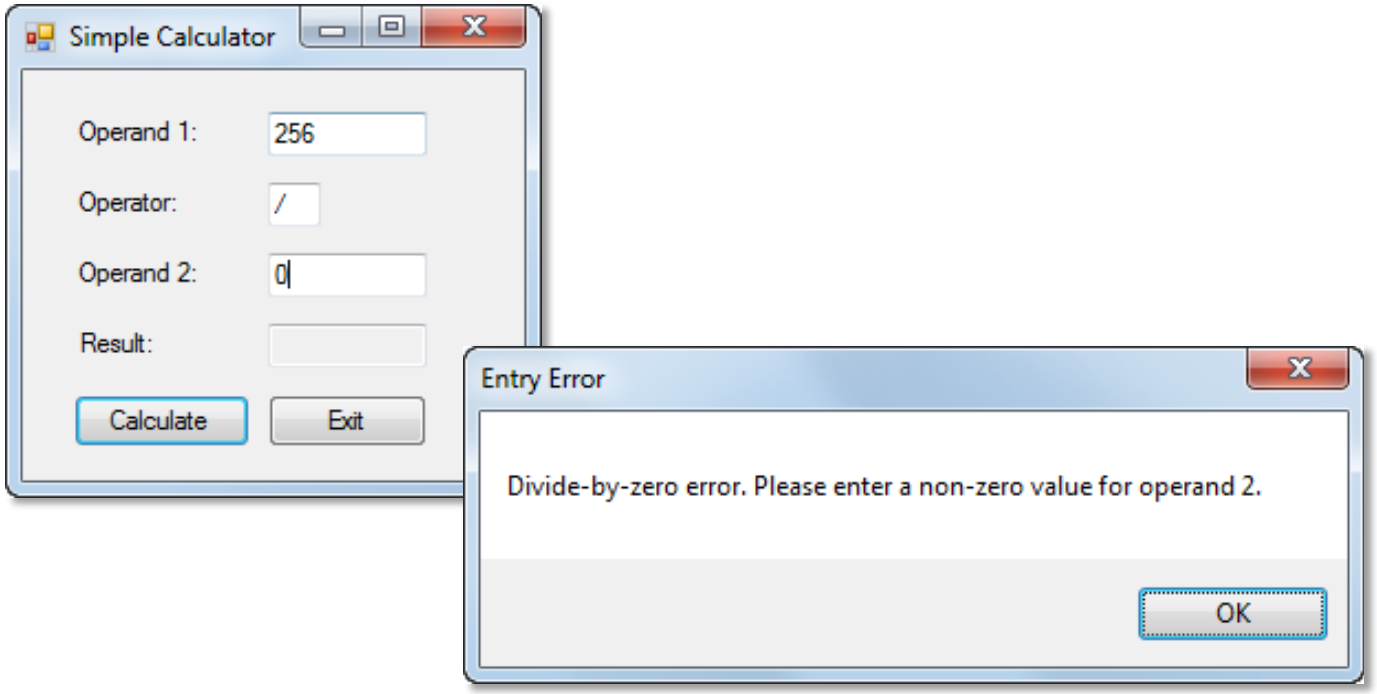
The code for the Future Value application (cont.)

```
public bool IsDecimal(TextBox textBox, string name)
{
    decimal number = 0m;
    if (Decimal.TryParse(textBox.Text, out number))
    {
        return true;
    }
    else
    {
        MessageBox.Show(name + " must be a decimal value.", "Entry Error");
        textBox.Focus();
        return false;
    }
}
public bool IsInt32(TextBox textBox, string name)
{
    int number = 0;
    if (Int32.TryParse(textBox.Text, out number))
    {
        return true;
    }
    else
    {
        MessageBox.Show(name + " must be an integer.", "Entry Error");
        textBox.Focus();
        return false;
    }
}
```

The code for the Future Value application (cont.)

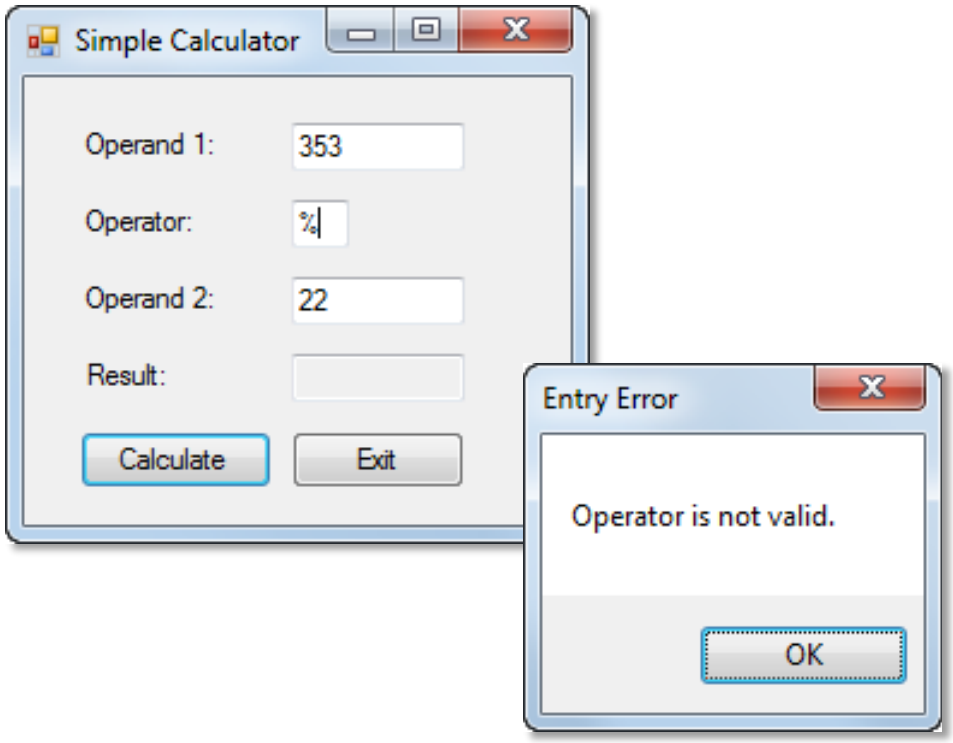
```
public bool IsWithinRange(TextBox textBox, string name,
    decimal min, decimal max)
{
    decimal number = Convert.ToDecimal(textBox.Text);
    if (number < min || number > max)
    {
        MessageBox.Show(name + " must be between " + min
            + " and " + max + ".", "Entry Error");
        textBox.Focus();
        return false;
    }
    return true;
}

private decimal CalculateFutureValue(decimal monthlyInvestment,
    decimal monthlyInterestRate, int months)
{
    decimal futureValue = 0m;
    for (int i = 0; i < months; i++)
    {
        futureValue = (futureValue + monthlyInvestment)
            * (1 + monthlyInterestRate);
    }
    return futureValue;
}
```

Extra 7-1 Add exception handling to the simple calculator



Add exception handling to the Simple Calculator form.

Extra 7-2 Add data validation to the simple calculator



Add data validation to the Simple Calculator form.