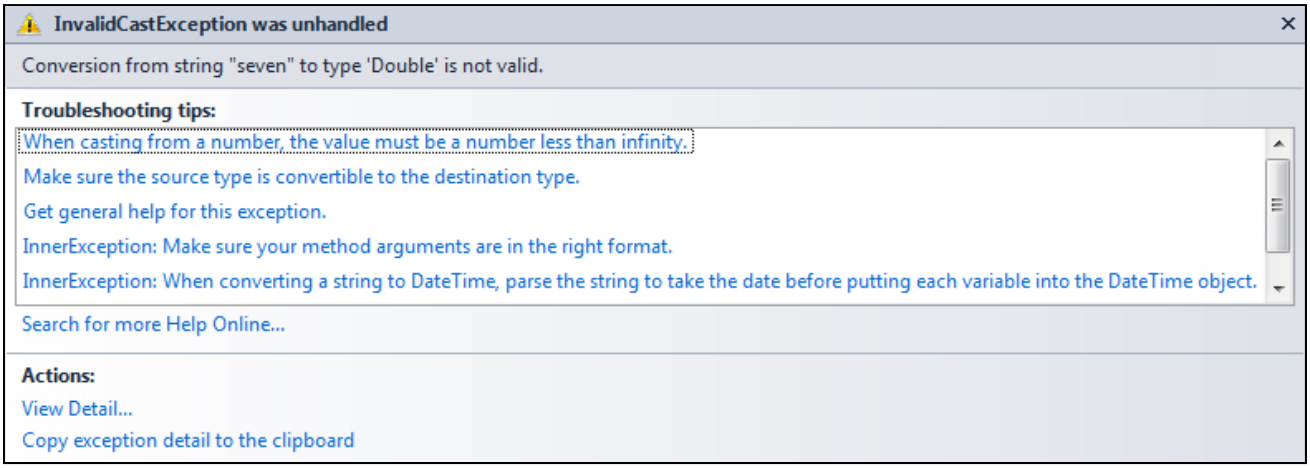
HANDLING EXCEPTIONS

Up to now we haven't dealt with what happens when a problem occurs during a program's execution. **Exception handling** is the process of creating applications that can resolve problems that may prevent a program from continuing normal execution. Let's say, for example, that a program requires an integer from the user, and instead the user enters a String value or a Double value. The programs that we have created up to this point would crash if the user didn't enter the values that were expected from the user. Instead of allowing our programs to crash, we are going to handle these types of issues by notifying the user of the issue and providing them with the opportunity to rectify the problem.

Let's go back to our **Parking Meter** program and reflect back on what happens when users do not enter a number in the text box where they are supposed to enter the number of hours they have been parked. When users do not enter a number in the textbox, the program crashes and Visual Basic outputs the following error message:



The line of code that generates the error is where the program tries to convert the text that the user enters into the text box into a Double:

```
hours = CDbl(txtHours.Text)
```

Instead of allowing the program to crash, it is important for a programmer to handle such errors. The way such errors can be handled is by using what is called a **Try-Catch** statement. A Try-Catch statement takes the following form:

```
Try

Catch ex As Exception

End Try
```

The **Try** block is where we instruct the program to try to execute the code. So, in the above example, we want to try to convert the text the user enters into the text box into a Double and store it in our variable.

The **Catch** block is where we handle any errors that may result in the execution of the code. Since trying to convert text into a Double throws an **InvalidCastException** error, we need to declare a variable (here called **ex**) as an **InvalidCastException** object.

So our **Try-Catch** statement for the Parking Meter program will look something like this:

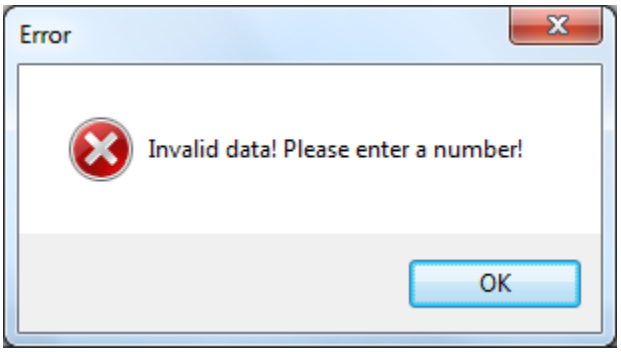
```
Try
   hours = CDbl(txtHours.Text)

   If hours < 1 And hours > 0 Then
      total = 3.0
   ElseIf hours >= 1 And hours <= 2 Then
      total = 5.0
   ElseIf hours > 2 And hours <= 3 Then
      total = 7.0
   ElseIf hours > 3 And hours <= 5 Then
      total = 10.0
   ElseIf hours > 5 Then
      total = 15.0
   Else
      hours = 0.0
   End If

   lblTotal.Text = FormatCurrency(total)

Catch ex As InvalidCastException
   MsgBox("Invalid data! Please enter a number!",
      MsgBoxStyle.Critical, "Error")
   txtHours.Focus()
   txtHours.SelectAll()
End Try
```

You'll notice that within the **Try** block, you need to include all the code that you want the program to try to execute. If the program is not able to convert the text the user enters in the text box into a Double, the program will skip the next lines of code within the **Try** block and execute the instructions in the **Catch** block. This time, instead of the program crashing, a message box will appear telling users that the information is invalid and allows users to enter a new value.



VALIDATING USER INPUT USING THE *Validating* EVENT PROCEDURE

Instead of validating the data users enter in the text box when users click OK, another way of validating input is by using a property named **CausesValidation** which is a property included in most VB controls. Controls that have a **CausesValidation** property that is set to **True** is capable of triggering a **Validating** event. A control's **Validating** event is triggered just before the focus is shifted to another control whose **CausesValidation** property is set to **True**.

So instead of waiting for users to click OK to validate their input, we are going to validate their input as soon as the user enters text in the text box and focus is removed from the textbox. In order to do this we need to write our **Try-Catch** statement within the **Validating** event procedure of the text box that we have called **txtHours**.

```
Private Sub txtHours_Validating(ByVal sender As Object, ByVal e As
System.ComponentModel.CancelEventArgs) Handles txtHours.Validating

   Try
      hours = CDbl(txtHours.Text)
   Catch ex As InvalidCastException
      MsgBox("Invalid data! Please enter a number!",
         MsgBoxStyle.Critical, "Error")
      txtHours.Focus()
      txtHours.SelectAll()
   End Try

End Sub
```

If the user enters, a value that is not a number in the text box, an error message will appear and the focus will return to the text box that was validated. The same will happen when the user clicks on the OK button.

However, we don't want the program to validate the data that is in the textbox when the user clicks **CLEAR** or **EXIT**. So in order to ensure that does not happen you will need to set the **CausesValidation** property for both the **CLEAR** and **EXIT** buttons to **False**.

After the **Validating** event has been triggered and the focus has shifted to another control, the **Validated** event is triggered. If you need to perform an operation on the user's input after it has been validated, such as storing the value in to a variable, you can write a **Validated** event procedure to do so.

For our purposes, if the data that the user enters is valid all we want to do is store the value the user enters into the texbox into our variable.

The calculation of the parking rate would continue to be calculated in the **Click** procedure of the **OK** button:

```
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnOK.Click

   Dim total As Double

   If hours < 1 And hours > 0 Then
      total = 3.0
   ElseIf hours >= 1 And hours <= 2 Then
      total = 5.0
   ElseIf hours > 2 And hours <= 3 Then
      total = 7.0
   ElseIf hours > 3 And hours <= 5 Then
      total = 10.0
   ElseIf hours > 5 Then
      total = 15.0
   Else
      hours = 0.0
   End If

   lblTotal.Text = FormatCurrency(total)

End Sub
```