Chapter 8 – Files

PRELIMINARIES

```
Private Sub Form1_Load() Handles MyBase.Load

 'read the file into an array. The file is assumed to be comma-delimited
 'Delaware, DE, 1954, 759000
 'Pennsylvania, PA, 44817, 12296000
 'New Jersey, NJ, 7417, 8135000

 Dim States() As String = IO.File.ReadAllLines("File.txt")

 'go through the array
 For i = 0 To States.GetUpperBound(0)
   Dim State() As String
   State = States(i).Split(",")

   'do something with State(0)
 Next
  End Sub
```

RealAllLines

- Read *all* the lines of a text-file into an array
  - Method opens a file
  - Reads each line of the file
  - Adds each line as an element of a string array
  - Closes the file

- A line is defined as a sequence of characters followed
  - carriage return
  - a line feed
  - a carriage return followed by a line feed

WriteAllLines

```
IO.File.WriteAllLines ("fileName.txt", States)
```

- Creates a new text file
- Copies the contents of a string array
- Places one element on each line
- Close the file

Set Operations

- Concat
  - Contains elements of array1 and array2
  - Duplication is OK

```
Dim States1() As String = {"A", "B", "C", "D"}
Dim States2() As String = {"A", "B", "G", "H"}

Dim States3() As String = _
                    States1.Concat(States2).ToArray()
```

Set Operations

- Union
  - Contains elements of array1 and array2
  - No Duplication

- `Dim States1() As String = {"A", "B", "C", "D"}`
- `Dim States2() As String = {"A", "B", "G", "H"}`

-
```
Dim States3() As String = _
                    States1.Union(States2).ToArray()
```

Set Operations

- Intersect
  - Contains elements from array1 and array2 which exist in *both* array1 and array2

- `Dim States1() As String = {"A", "B", "C", "D"}`
- `Dim States2() As String = {"A", "B", "G", "H"}`

- `Dim States3() As String = _`

```
    States1.Intersect(States2).ToArray()
```

Set Operations

- Except
  - Contains elements from array1 which do not exist in array2

- `Dim States1() As String = {"A", "B", "C", "D"}`
- `Dim States2() As String = {"A", "B", "G", "H"}`

-
```
Dim States3() As String = _
                    States1.Except(States2).ToArray()
```

Opening a File

- Add *OpenFileDialog* control to form
- To show the Open dialog box
  - `OpenFileDialog1.ShowDialog()`
- After selecting the file, it'll be stored in
  - `OpenFileDialog1.`*`FileName`*

Sequential Files

Sequential Files

- A sequential file consists of data stored in a text file on disk.
- May be created with Visual Studio
- May also be created programmatically from Visual Basic

Creating a Sequential File

1. Choose a filename – may contain up to 215 characters

2. Select the path for the folder to contain this file

3. Execute a statement like the following:

```
Dim sw As IO.StreamWriter =
    IO.File.CreateText(filespec)
```

(Opens a file for output.)

Creating a Sequential File...

4. Place lines of data into the file with statements of the form:

```
sw.WriteLine(datum)
```

5. Close the file:

```
sw.Close()
```

Note: If no path is given for the file, it will be placed in the *Debug* subfolder of *bin*.

EXAMPLE

```
Private Sub btnCreateFile_Click(...) _
  Handles btnCreateFile.Click
 Dim sw As IO.StreamWriter =
IO.File.CreateText("PAYROLL.TXT")
  sw.WriteLine("Mike Jones") 'Name
  sw.WriteLine(9.35)  'Wage
  sw.WriteLine(35)    `Hours worked
  sw.WriteLine("John Smith")
  sw.WriteLine(10.75)
  sw.WriteLine(33)
  sw.Close()
End Sub
```

File: PAYROLL.TXT

Mike Jones
9.35
35
John Smith
10.75
33

CAUTION

- With **`IO.File.CreateText`**
  - If an existing file is opened for output, Visual Basic will erase the existing file and create a new one.

Adding Items to a Sequential File

1. Execute the statement

```
Dim sw As IO.StreamWriter =
IO.File.AppendText(filespec)
```

where *sw* is a variable name and *filespec* identifies the file.

2. Place data into the file with the **`WriteLine`** method.

3. After all the data have been recorded into the file, close the file with the statement

```
sw.Close()
```

IO.File.AppendText

- Will add data to the end of an existing file
- If a file does not exist, the method will create it

Sequential File Modes

- CreateText – open for output
- OpenText – open for input
- AppendText – open for append
- A file should not be opened in two different modes at the same time.

Avoiding Errors

- Attempting to open a non-existent file for input brings up a message box titled:

  `FileNotFoundException`

- There is a method to determine if a file exists before attempting to open it:

  `IO.File.Exists(filespec)`

  will return a True if the file exists

Testing for the Existence of a File

```
Dim sr As IO.StreamReader
If IO.File.Exists(filespec) Then
 sr = IO.File.OpenText(filespec)
Else
 message = "Either no file has yet been "
 message &= "created or the file named"
 message &= filespec & " is not found."
 MessageBox.Show(message, "File Not Found")
End If
```

Deleting Info from a Sequential File

- An individual item of a file cannot be changed or deleted directly
- A new file must be created by reading each item from the original file and recording it, with the single item changed or deleted, into the new file
- The old file is then erased, and the new file renamed with the name of the original file

Delete and Move Methods

- Delete method:

```
IO.File.Delete(filespec)
```

- Move method (to change the filespec of a file):

```
IO.File.Move(oldfilespec,
             newfilespec)
```

- ***Note:*** The IO.File.Delete and IO.File.Move methods cannot be used with open files.

Imports System.IO

- Simplifies programs that have extensive file handling

- Place the statement

```
Imports System.IO
```

at the top of the Code Editor, before the `Class frmName` statement. Then, there is no need to insert the prefix "IO." before the words StreamReader, StreamWriter, and File

Exception Handling

Structured Exception Handling

- Two types of problems in code:
  - *Bugs* (logic error) – something wrong with the code the programmer has written
  - *Exceptions* – errors beyond the control of the programmer

- Programmer can use the debugger to find bugs; but must anticipate exceptions in order to be able to keep the program from terminating abruptly.

How Visual Basic Handles Exceptions

- An unexpected problem causes Visual Basic first to throw an exception then to handle it
- If the programmer does not explicitly include exception-handling code in the program, then Visual Basic handles an exception with a default handler
- The default exception handler terminates execution, displays the exception's message in a dialog box and highlights the line of code where the exception occurred

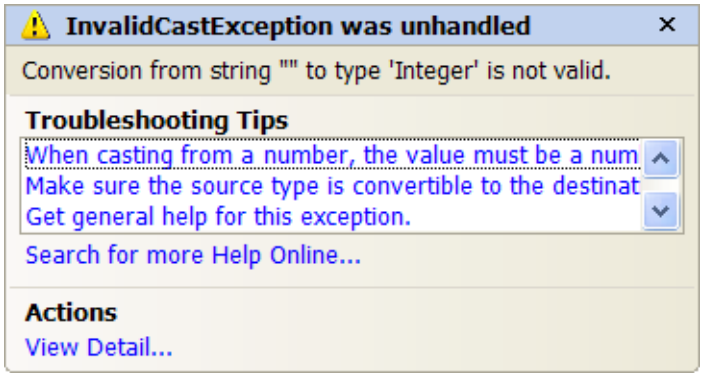
Exception Example

- If the user enters a word or leaves the input box blank in the following program, an exception will be thrown:

```
Dim taxCredit As Double

Private Sub btnComputeCredit_Click(...) _
    Handles btnComputeCredit.Click
 Dim numDependants As Integer
 numDependants = CInt(InputBox( _
"How many dependants do you have?"))
 taxCredit = 1000 * numDependants
End Sub
```

Exception Handled by Visual Basic



Try-Catch-Finally Block

```
Dim taxCredit As Double
Private Sub btnComputeCredit_Click(...)
 Handles btnComputeCredit.Click
 Dim numDependents As Integer, message As String
 Try
  numDependents = CInt(InputBox("How many
  dependents?"))
 Catch
  message = "You did not answer the question " _
   & " with an integer value. We will " _
   & " assume your answer is zero."
  MessageBox.Show(message)
  numDependents = 0
 Finally
  taxCredit = 1000 * numDependents
 End Try
End Sub
```

Catch Blocks

- Visual Basic allows Try-Catch-Finally blocks to have one or more specialized Catch clauses that only trap a specific type of exception.

- The general form of a specialized Catch clause is

```
Catch exp As ExceptionName
```

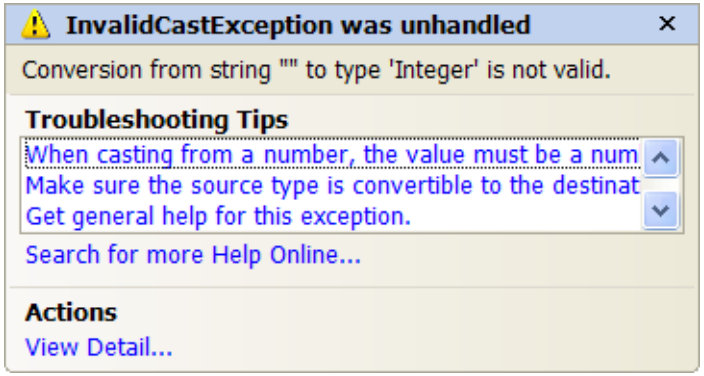
- where the variable **exp** will be assigned the name of the exception. The code in this block will be executed only when the specified exception occurs.

Try Catch Block Syntax

```
Try
 normal code
Catch exc1 As FirstException
 exception-handling code for FirstException
Catch exc2 As SecondException
 exception-handling code for SecondException
.
.
Catch
 exception-handling code for any remaining
  exceptions
Finally
 clean-up code
End Try
```

Example Error Handling

```
Dim x As Integer = 0
Dim div As Integer = 0
Try
  div = 100 / x
  Console.WriteLine("Not executed line")
Catch de As DivideByZeroException
  Console.WriteLine("DivideByZeroException")
Catch ee As Exception
  Console.WriteLine("Exception")
Finally
  Console.WriteLine("Finally Block")
End Try
Console.WriteLine("Result is {0}", div)
```

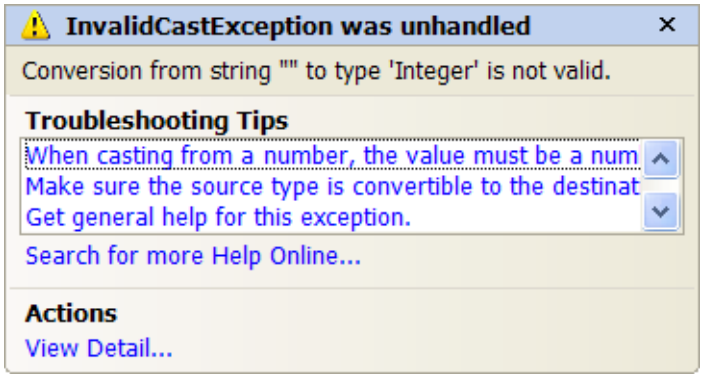


Example Error Handling

```
Dim x As Integer = 0
    Dim div As Integer = 0
    Try
      div = 100 / x
      Console.WriteLine("Not executed line")
    Catch de As Exception

      If de.Message = "Arithmetic operation resulted in an overflow."  Then
        Console.WriteLine("Overflow")
      Else
        Console.WriteLine("DivideByZeroException")
      End If

    Finally
      Console.WriteLine("Finally Block")
    End Try
    Console.WriteLine("Result is {0}", div)
```



Exception Handling and File Errors

- Exception handling can also catch file access errors.
- File doesn't exist causes an IO.FileNotFoundException
- If an attempt is made to delete an open file, IO.IOException is thrown.

Practice

```
Private Sub btnDisplay_Click(...) Handles btnDisplay.Click
        Try
            Dim caPop As Integer = 3405500
            Dim worldPop As Integer
            worldPop = 1970 * caPop
            txtOutput.Text = CStr(worldPop)

        Catch ex As ArgumentOutOfRangeException
            txtOutput.Text = "Oops"

        Catch exe As OverflowException
            txtOutput.Text = "Error occurred."
        End Try

    End Sub
```

Using Sequential Files

CSV File Format

- Comma Separated Values
- Records are stored on one line with a comma between each field
- *Example:*

Mike Jones,9.35,35

John Smith,10.75,33

LSV File Format

- Line Separated Values
- Each value appears on its own line
- Up to now, this is the only type of file we have been using

Split Example

- For instance, suppose the String array employees() has been declared without an upper bound, and the String variable *line* has the value "Bob,23.50,45".

```
employees = line.Split(","c)
```

- sets the size of employees() to 3
- sets employees(0) = "Bob"
- employees (1) = "23.50"
- employees(2) = "45".

Split Comments

```
Employees = line.Split(","c)
```

- In this example, the character comma is called the **delimiter** for the Split function, and the letter *c* specifies that the comma has data type Character instead of String. (If Option Strict is Off, the letter *c* can be omitted.)

- Any character can be used as a delimiter. If no character is specified, the Split function will use the space character as delimiter.

EXAMPLE 2

```
Private Sub btnConvert_Click(...) _
  Handles btnConvert.Click
 Dim stateData(), line As String
 line = "California, 1850, Sacramento, Eureka"
 stateData = line.Split(","c)
 For i As Integer = 0 To
  stateData.GetUpperBound(0)
  stateData(i) = stateData(i).Trim 'Get rid
 'of extraneous spaces
  lstOutput.Items.Add(stateData(i))
 Next
End Sub
```

California
1850
Sacramento
Eureka

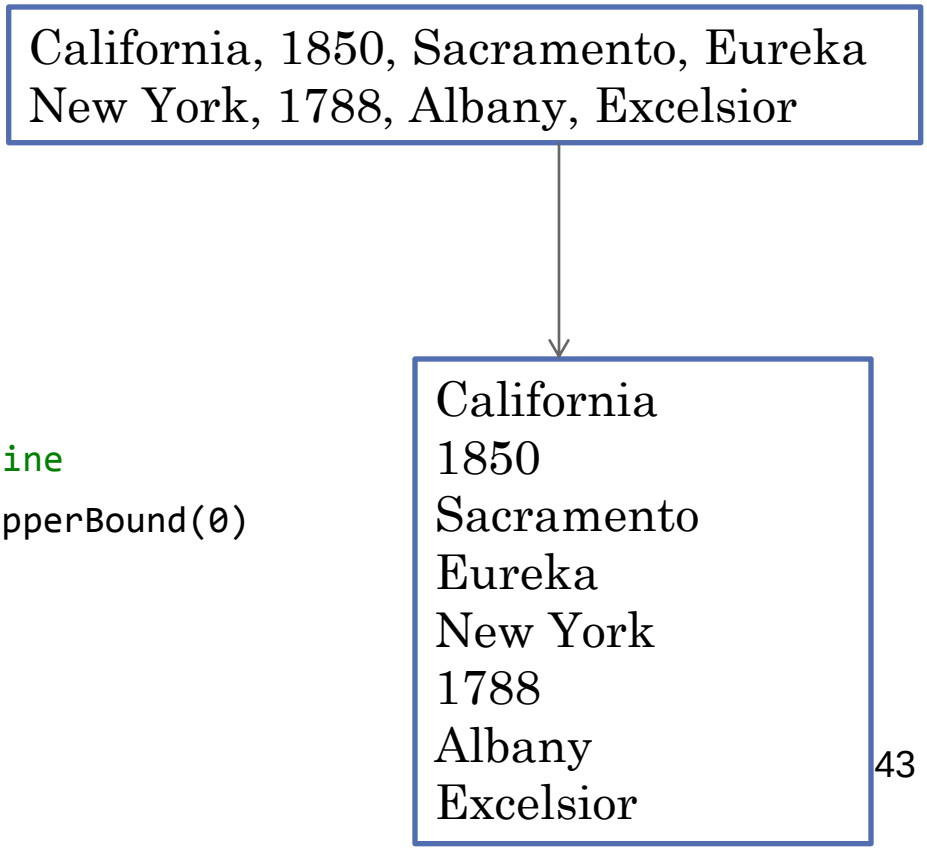
Example 3: Convert a CSV Format File to an LSV Format

```
'loop until there's something in the file
Do While (sr.Peek() <> -1)

  'read a single line
  line = sr.ReadLine()

  'take the fields out of the line
  fields = line.Split(","c)

  'write each field onto a seperate line
  For i As Integer = 0 To fields.GetUpperBound(0)
    sw.WriteLine(fields(i).Trim)
  Next
Loop
```



JOIN FUNCTION

- The reverse of the Split function is the Join function

- Join concatenates the elements of a string array into a string containing the elements separated by a specified delimiter.

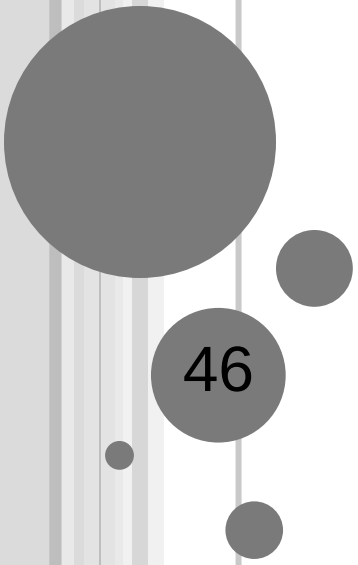
```
Dim greatLakes() As String = _
    {"Huron","Ontario","Michigan","Erie","Superior"}
Dim lakes As String
lakes = String.Join(",", greatLakes)
txtOutput.Text = lakes
```

OUTPUT:

```
Huron,Ontario,Michigan,Erie,Superior
```

Comments

- Files to be processed can be opened and closed within a single procedure.
- Files can also be opened just once the instant the program is run and stay open until the program is terminated.
- To open a file once, open it in the form's Load procedure and put the Close method and End statement in the click event procedure for a button labeled "Quit."



HashTables

Working with HashTable

- Collection
  - a set of objects that can be access by iterating through each element in turn

- So?
- Even an array can hold a set of objects

Working with HashTable

- more flexibility when using a collection object when compared to arrays

- Arrays are of fixed size
- for a Collection we can keep on adding elements to it

Working with HashTable

- Arrays can store only one data type
- collections can hold any objects
- Accessing the element is very simple and very fast
- Removing the element in Collection is very simple

Working with HashTable

- Array:
  - MyArray(100) returns element 100
- What if I want element “George”?
- Hashtable
  - MyHash.Item(“George”)

Working with HashTable

- **Declaring HashTable**

- Dim MyHash As New Hashtable

Working with HashTable

- **Adding an element to the HashTable**

- {hash table object}.Add(Key as Object, value as Object)

- Ex: MyHash.Add("George", 45)

Working with HashTable

- **Accessing an element**

  {hash table object}.Item({key})

  Ex: MyArray.Item("George")

Working with HashTable

- **Searching for an element**

- {hash table object}.Contains({key})

  Ex: MyArray.Contains("George")