



# Sequential Files

---

November 3, 2006

# Chapter 8 – Sequential Files

---

## 8.1 Sequential Files

## 8.2 Using Sequential Files

# Section 8.1 – Sequential Files

---

- Creating a Sequential File
- Adding Items to a Sequential File
- Structured Exception Handling

# Sequential Files

---

- A sequential file consists of data stored in a text file on disk.
- May be created using Notepad
- May also be created directly 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(7.35)            'Wage  
    sw.WriteLine(35)              'Hours worked  
    sw.WriteLine("John Smith")  
    sw.WriteLine(6.75)  
    sw.WriteLine(33)  
    sw.Close()  
End Sub
```

# File: PAYROLL.TXT

---

Mike Jones

7.35

35

John Smith

6.75

33



# Caution

---

- 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, it 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."
    MsgBox(message, 0, "File Not Found")
End If
```

# Deleting Information 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**

near the top of the Code window, before the Class frmName statement. Then, there is no need to insert the prefix “IO.” before the words StreamReader, StreamWriter, and File.

# Structured Exception Handling

---

- Two types of problems in code:
  - *Bugs* – 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

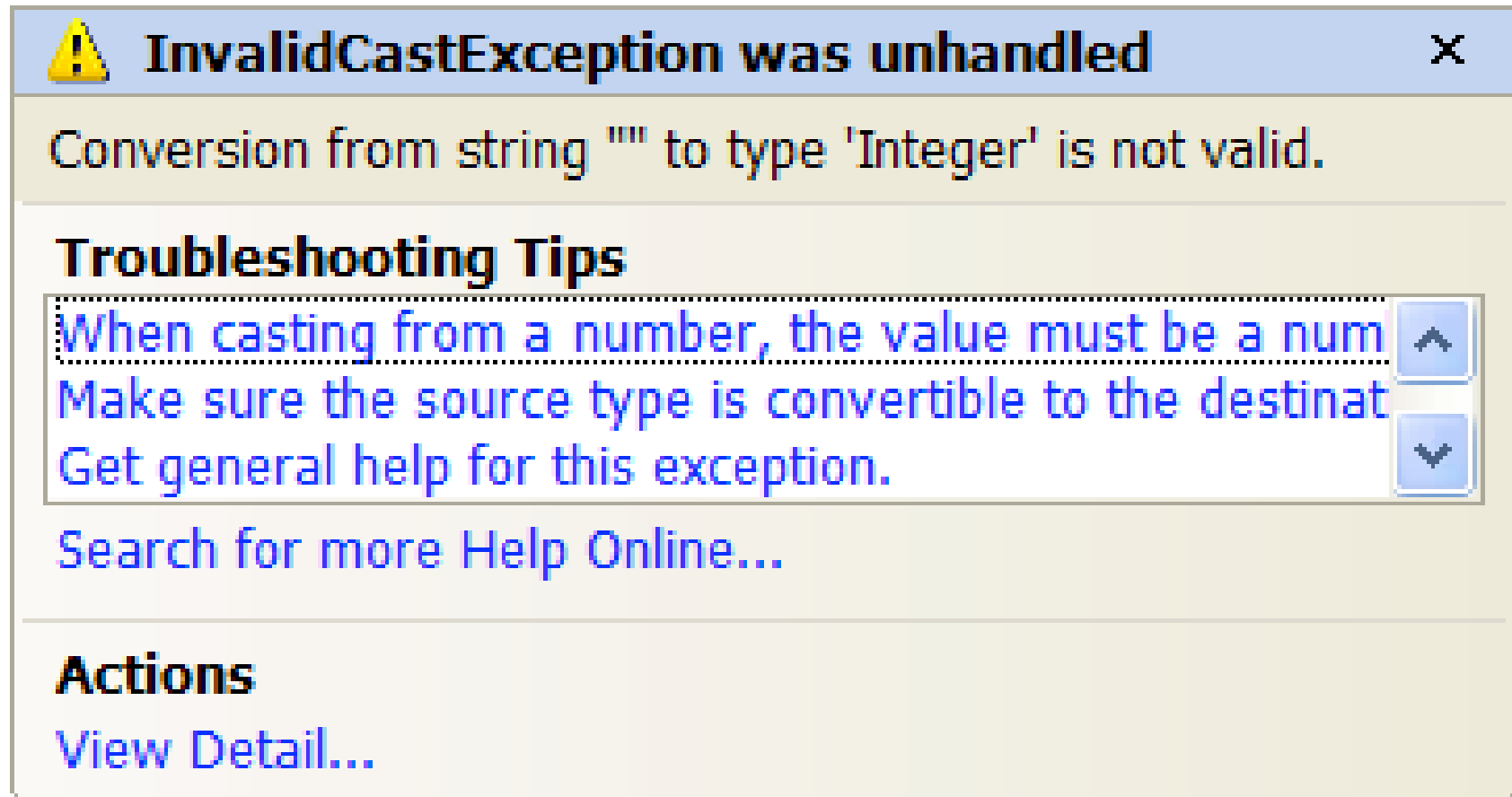
---

- 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(TextBox( _
        "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."
        MsgBox(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



# 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.

## 8.2 Using Sequential Files

---

- Sorting Sequential Files
- CSV Format
- Merging Sequential Files
- Control Break Processing

## 8.2 Sorting Sequential Files

---

1. Read data from file into an array of structures.
2. Sort the data based on chosen member in structure.
3. Write sorted data to file.

# CSV File Format

---

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

Mike Jones,7.35,35

John Smith,6.75,33

# 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 Function

---

- Facilitates working with CSV formatted files.
- Split can convert a line containing commas into a String array.
- The 0th element contains the text preceding the first comma, the 1st element contains the text between the first and second commas, ..., and the last element contains the text following the last comma.

# 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(",")  
    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
```

# Example 2 Output

---

California

1850

Sacramento

Eureka

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

---

```
Private Sub btnConvert_Click(...) Handles btnConvert.Click
    Dim line, fields(), fromFile, toFile As String
    Dim sr As IO.StreamReader
    Dim sw As IO.StreamWriter
    fromFile = InputBox("Name of original file:", _
        "Convert from CSV to LSV")
    toFile = InputBox("Name of converted file:", _
        "Convert from CSV to LSV")
    sr = IO.File.OpenText(fromFile)
    sw = IO.File.CreateText(toFile)
    Do While (sr.Peek() <> -1)
        line = sr.ReadLine()
        fields = line.Split(", "c)
```

## Example 3 continued

---

```
    For i As Integer = 0 To fields.GetUpperBound(0)
        sw.WriteLine(fields(i).Trim)
    Next
Loop
sr.Close()
sw.Close()
sr = IO.File.OpenText(toFile)
Do While sr.Peek <> -1
    lstFile.Items.Add(sr.ReadLine)
Loop
sr.Close()
End Sub
```

## Example 3: Input File

---

California, 1850, Sacramento, Eureka  
New York, 1788, Albany, Excelsior

# Example 3: Output File

---

California

1850

Sacramento

Eureka

New York

1788

Albany

Excelsior

# 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 = Join(greatLakes, ",")  
txtOutput.Text = lakes
```

*OUTPUT:*

Huron,Ontario,Michigan,Erie,Superior

# Merging Ordered Sequential Files Algorithm

---

1. Open the two ordered files for input, and open a third file for output.
2. Try to get an item of data from each file.
3. Repeat the following steps until an item of data is not available in one of the files:
  - a) If one item precedes the other, write it into the third file and try to get another item of data from its file.
  - b) If the two items are identical, write one into the third file and try to get another item of data from each of the two ordered files.



## Merge Algorithm continued

---

4. At this point, an item of data has most likely been retrieved from one of the files and not yet written to the third file. In this case, write that item and all remaining items in that file to the third file.
5. Close the three files.

# Control Break Processing

---

- Used to create subtotals
- When there is special significance to the changing of the value of a certain variable, that variable is called a **control variable**
- Each change of its value is called a **break**.

## Data for Example 5

---

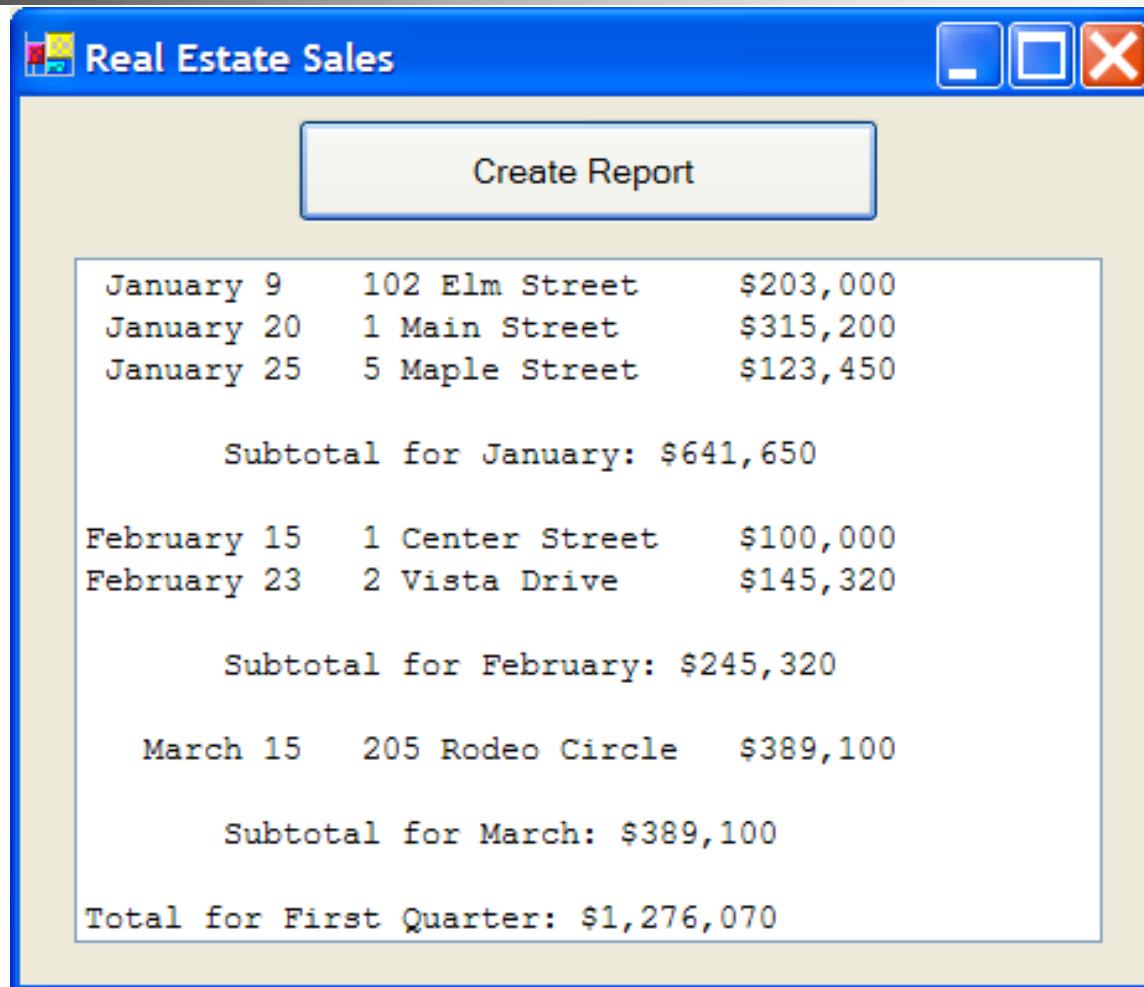
<u>Month</u>	<u>Day</u>	<u>Address</u>	<u>Price</u>
January	9	102 Elm St	\$203,000
January	20	1 Main St	\$315,200
January	25	5 Maple St	\$123,450
February	15	1 Center St	\$100,000
February	23	2 Vista Dr	\$145,320
March	15	5 Rodeo Cir	\$389,100

# Task for Example 5

---

- Display sales by month and display the monthly subtotals.

# Output for Example 5



The screenshot shows a Windows application window titled "Real Estate Sales". Inside the window, there is a button labeled "Create Report". Below the button is a text area containing the following data:

January 9	102 Elm Street	\$203,000
January 20	1 Main Street	\$315,200
January 25	5 Maple Street	\$123,450
Subtotal for January:		\$641,650
February 15	1 Center Street	\$100,000
February 23	2 Vista Drive	\$145,320
Subtotal for February:		\$245,320
March 15	205 Rodeo Circle	\$389,100
Subtotal for March:		\$389,100
Total for First Quarter:		\$1,276,070

# 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."