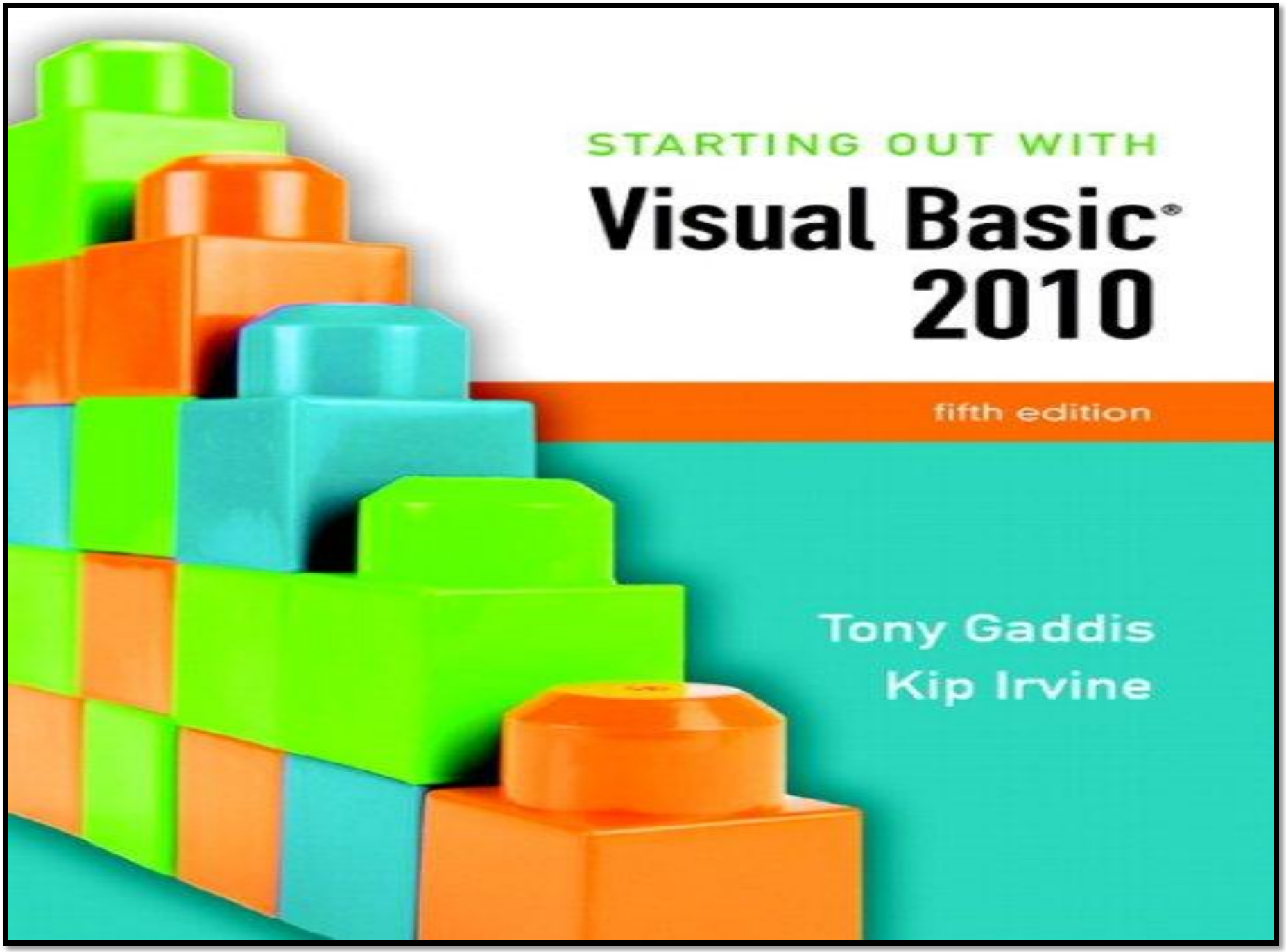
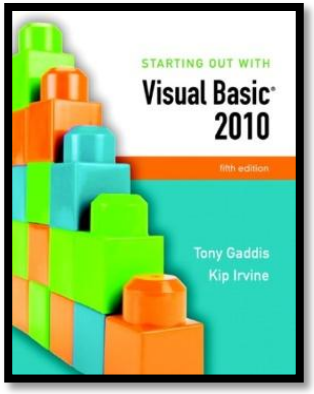

STARTING OUT WITH
Visual Basic®
2010
fifth edition
Tony Gaddis
Kip Irvine



Chapter 9

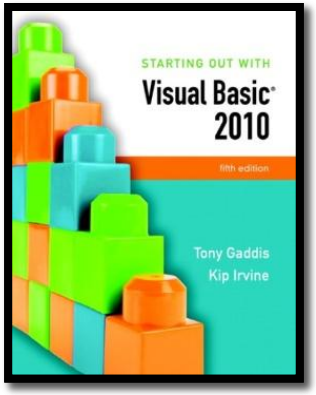
Files, Printing, and Structures

Addison Wesley
is an imprint of

PEARSON

Introduction

- In this chapter you will learn how to:
  - Save data to sequential text files
  - Read data from the files back into the application
  - Use the OpenFileDialog, SaveFileDialog, ColorDialog, and FontDialog controls
    - For opening and saving files and for selecting colors and fonts with standard Windows dialog boxes
  - Use the PrintDocument control
    - To print reports from your application
  - Package units of data together into structures



Section 9.1

USING FILES

A file is a collection of data stored on a computer disk.

Data can be saved in a file and later reused.

Addison Wesley
is an imprint of

PEARSON

Data Can be Stored in a File

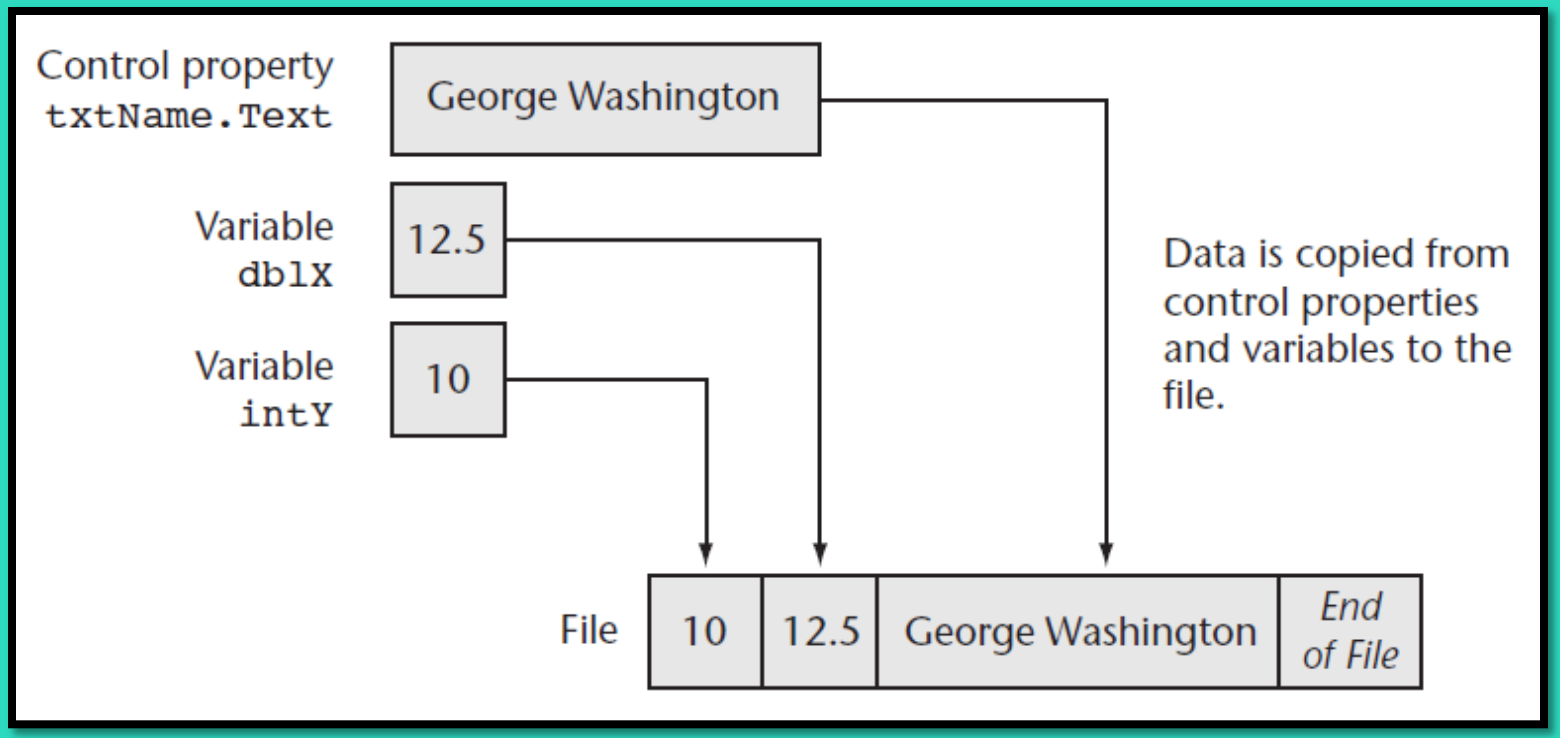
- Thus far, all of our data has been stored in controls and variables existing in RAM
- This data disappears once the program stops running
- If data is stored in a **file** on a computer disk, it can be retrieved and used at a later time

The Process of Using a File

- The following steps must be taken when a file is used by an application:
  1. The file must be opened; If it does not yet exist, it must be created
  2. Data is written to the file or read from the file
  3. When the application is finished using the file, the file is closed

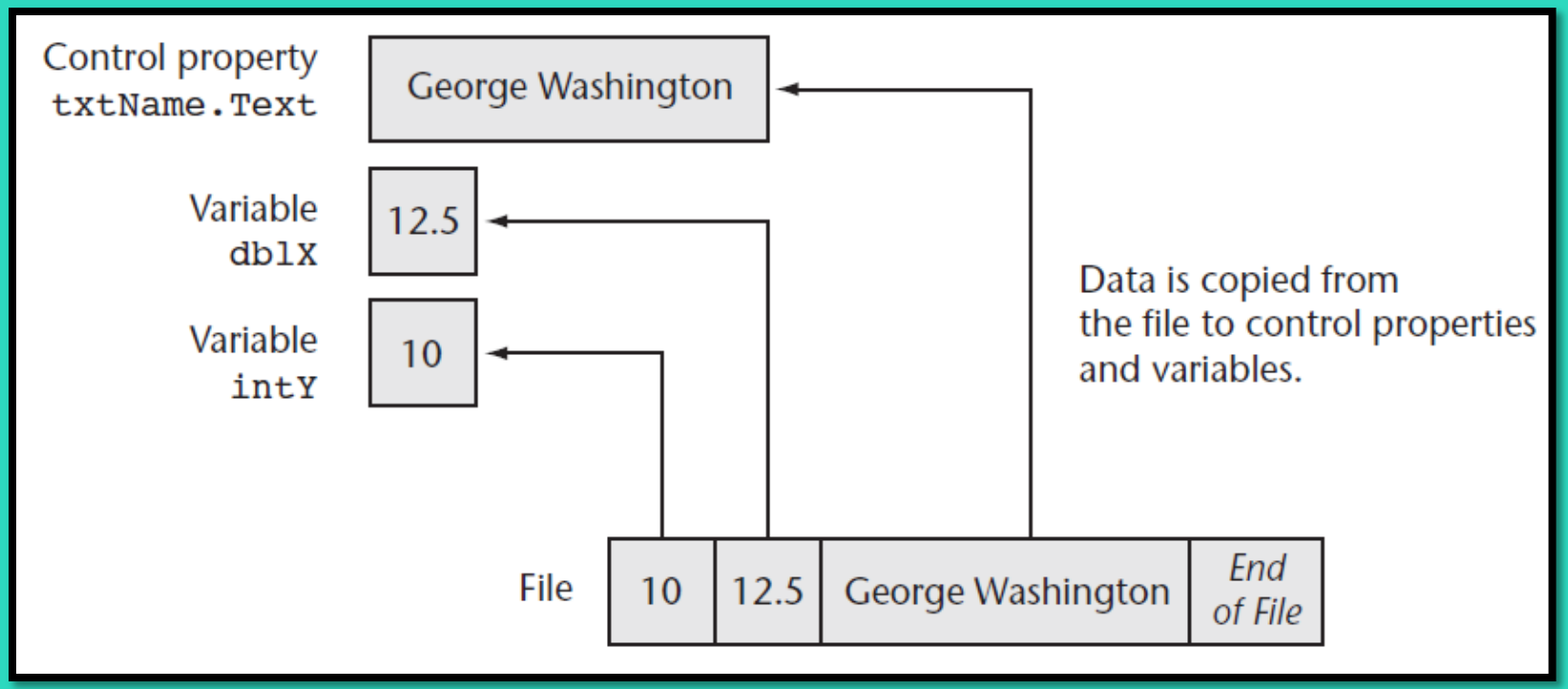
Output File

- An **output file** is a file into which a program writes data



Input File

- An **input file** is a file from which a program reads data



File Types

- There are two types of files:
  - Text
  - Binary
- A **text file** contains plain text and may be opened in a text editor such as Windows Notepad
- **Binary files** contain pure binary data and cannot usually be viewed with a text editor

File Access Methods

- There are two methods of accessing Files:
  - Sequential-access
  - Random-access
- A **sequential-access** file is like a stream of data that must be read from beginning to end
- A **random-access** file may be accessed in any order

Writing to Files with **StreamWriter** Objects

- Two basic ways to open a file for writing
  - Create a new file
  - Open an existing file and append data to it
- A **StreamWriter object** performs the actual writing to the file
- Two required steps:
  - Declare a **StreamWriter** variable
  - Call either **File.CreateText** or **File.AppendText** and assign its return value to the **StreamWriter** variable

Using the **Imports** Statement for the **StreamWriter** Classes

- To make the **StreamWriter classes** available to your program
  - Insert the following **Imports** statement at the top of your form's code file:

Imports System.IO

NOTE: It is possible to omit the `Imports System.IO` statement, but then every reference to the `StreamWriter` class must use its fully qualified name, which is `System.IO.StreamWriter`.

Creating a Text File

- Declare a **StreamWriter** variable using the following general format:

  Dim *ObjectVar As StreamWriter*

  - ***ObjectVar*** is the name of the object variable
  - You may use **Private** or **Public** in place of **Dim**
    - At the class-level or module-level
  - Here's an example:

  Dim phoneFile As StreamWriter

Creating a Text File

- Next, call the **File.CreateText method**, passing the name of a file
- For example:

```
phoneFile = File.CreateText("phonelist.txt")
```

  - Notice the return value from **File.CreateText** is assigned to the **StreamWriter** variable named **phoneFile**

File Paths

- The filename that you pass to the **File.CreateText** method
  - Can be a complete file path with drive letter

    "C:\data\vbfiles\phonelist.txt"
  - Refer to a file in the default drive root directory

    "\phonelist.txt"
  - Include no path information at all

    "phonelist.txt"
- If no path information specified
  - The **\bin\Debug** folder of the current project is used

Opening an Existing File and Appending Data to It

- If a text file already exists, you may want to add more data to the end of the file
  - This is called *appending* the file
- First, declare a **StreamWriter** variable
- Then call the **File.AppendText** method, passing the name of an existing file
  - If the file does not exit it will be created
- For example:

```
phoneFile = File.AppendText("phonelist.txt")
```

Writing Data to a File

- The **WriteLine method** of the **StreamWriter** class writes a line of data to a file using the following general format:

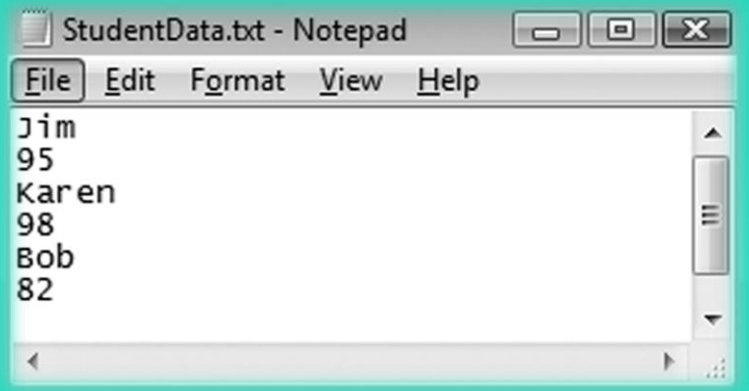
  ObjectVar.WriteLine(Data)

  - ***ObjectVar*** is the name of the **StreamWriter** object variable
  - ***Data*** represents constants or variables whose contents will be written to the file
    - Calling the method without the ***Data*** argument writes a blank line to the file
- The **WriteLine** method writes the data to the file and then writes a newline character immediately after the data
  - A **newline character** is an invisible character that separates text by breaking it into another line when displayed on the screen

Writing Data to a File

- The following writes three students' first names and scores to a file:

```
' Write data to the file.
studentFile.WriteLine("Jim")
studentFile.WriteLine(95)
studentFile.WriteLine("Karen")
studentFile.WriteLine(98)
studentFile.WriteLine("Bob")
studentFile.WriteLine(82)
```



Jim<*newline*>**95**<*newline*>**Karen**<*newline*>**98**<*newline*>**Bob**<*newline*>**82**<*newline*>

- In addition to separating the contents of a file into lines, the newline character also serves as a delimiter
  - A **delimiter** is an item that separates other items
  - Data must be separated in order for it to be read from a file

The **Write** Method

- The **Write method** is a member of the **StreamWriter** class that writes an item of data without writing a newline character using the following general format:

  ObjectVar.Write(Data)

  - ***ObjectVar*** is the name of a **StreamWriter** object
  - ***Data*** represents the contents of a constant or variable that is written to the file
  - Writes data to a file without terminating the line with a newline character
    - A blank space or comma could be used to provide a delimiter between data items

Closing a File

- The **StreamWriter** class has a method named **Close** that closes a file using the following general format:

  ObjectVar.Close()

  - ***ObjectVar*** is the **StreamWriter** object variable you want to close
    - The following statement closes a **StreamWriter** object variable named **salesFile**:

      salesFile.Close()
- The **Close method**
  - Writes any unsaved information remaining in the file **buffer**
  - Releases memory allocated by the **StreamWriter** object
- Tutorial 9-1 examines an application that writes data to a file

Appending a File

- When we **append** a file
  - We write new data immediately following existing data in the file
- If an existing file is opened with the **AppendText** method
  - Data written to the file is appended to the file's existing data
  - If the file does not exit, it is created

Appending a File Example

- The following example:

  Opens a file in append mode and writes additional data to the file

Before

```
Jim Weaver
555-1212
Mary Duncan
555-2323
Karen Warren
555-3434
```

```
' Declare an object variable
Dim friendFile As StreamWriter

' Open the file.
friendFile = File.AppendText("MyFriends.txt")

' Write the data.
friendFile.WriteLine("Bill Johnson")
friendFile.WriteLine("555–4545")

' Close the file.
friendFile.Close()
```

After

```
Jim Weaver
555-1212
Mary Duncan
555-2323
Karen Warren
555-3434
Bill Johnson
555-4545
```

Reading Files with **StreamReader** Objects

- A **StreamReader object** reads data from a sequential text file
  - A **StreamReader** object is an instance of the **StreamReader** class
- The **StreamReader class** provides methods for reading data from a file
- Create a **StreamReader** object variable using the following general format:

  Dim *ObjectVar As StreamReader*

  - ***ObjectVar*** is the name of the object variable
    - You may use **Private** or **Public** in place of **Dim**
      - At the class-level or module-level

Reading Files with **StreamReader** Objects

- The **File.OpenText method** opens a file and stores the address of the **StreamReader** object variable using the following general format:

  File.OpenText(*Filename*)

  - ***Filename*** is a string or a string variable specifying the path and/or name of the file to open
    - For example:

```
Dim customerFile As StreamReader
customerFile = File.OpenText("customers.txt")
```

- To make the **StreamReader** classes available
  - Write the following **Imports** statement at the top of your code file:

```
Imports System.IO
```

Reading Data from a File

- The **ReadLine method** in the **StreamReader** class reads a line of data from a file using the following general format:

  ObjectVar.ReadLine()

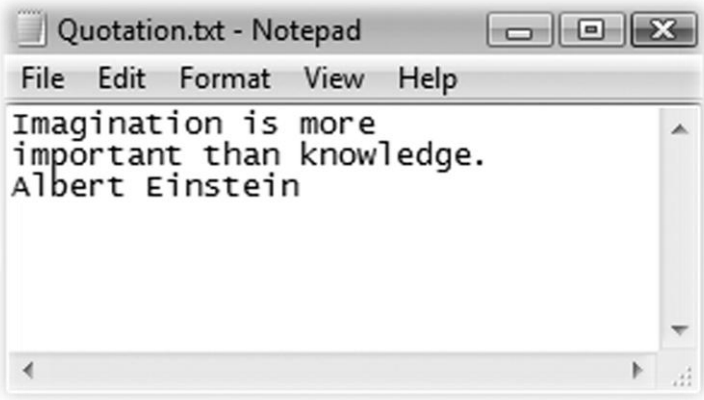
  - ***ObjectVar*** is the name of a **StreamReader** object variable
  - The method reads a line from the file associated with ***ObjectVar*** and returns the data as a string
    - For example, the following statement reads a line from the file and stores it in the variable:

      strCustomerName = customerFile.ReadLine()

Reading Data from a File

- Data is read from a file in a forward-only direction
- When the file is opened
  - Its **read position** is set to the first item in the file
- As data is read
  - The read position advances through the file

```
Dim textFile As StreamReader
textFile = File.OpenText("Quotation.txt")
```




```
strInput = textFile.ReadLine()
```

Closing the File

- The **StreamReader** class has a method named **Close** that closes an open **StreamReader** object using the following general format:

  ObjectVar.Close()

  - ***ObjectVar*** is the **StreamReader** object variable you want to close
    - The following statement closes a **StreamReader** object variable named **readFile**:

      readFile.Close()

- In Tutorial 9-2, you complete an application that uses the **ReadLine** statement

Determining Whether a File Exists

- To determine if a file exists before opening it, you can call the **File.Exists method** using the following general format:

  File.Exists(*Filename*)

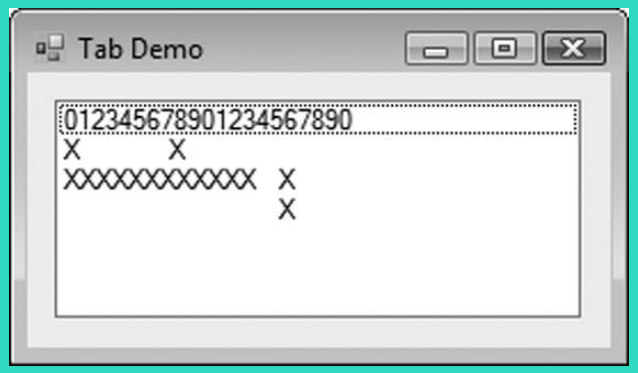
  - ***Filename*** is the name of a file, which may include the path
  - The method returns **True** if the files exists or **False** if the file does not exist

```
If File.Exists(strFilename) Then
   ' Open the file.
   inputFile = File.OpenText(strFilename)
Else
   MessageBox.Show(strFilename & " does not exist.")
End If
```

Using **vbTab** to Align Display Items

- The predefined **vbTab** constant
  - Moves the print position forward to the next even multiple of 8
  - Can be used to align columns in displayed or printed output

```
ListBox1.Items.Add("01234567890123456789")
ListBox1.Items.Add("X" & vbTab & "X")
ListBox1.Items.Add("XXXXXXXXXXXX" & vbTab & "X")
ListBox1.Items.Add(vbTab & vbTab & "X")
```



Detecting the End of a File

- In many cases, the amount of data in a file is unknown
- Use the **Peek method** to determine when the end of the file has been reached
- Here is the general format:

ObjectVar.Peek

  - ***ObjectVar*** is the name of a **StreamReader** object variable
  - The method looks ahead in the file without moving the read position
  - Returns the next character that will be read or **-1** if no more characters can be read

- The following example uses a **Do Until** loop and the **Peek** method to determine the end of the file:

```
Dim scoresFile As StreamReader
Dim strInput As String
scoresFile = File.OpenText("Scores.txt")
Do Until scoresFile.Peek = -1
   strInput = scoresFile.ReadLine()
   lstResults.Items.Add(strInput)
Loop
scoresFile.Close()
```

- Tutorial 9-3 examines an application that detects the end of a file

Other **StreamReader** Methods

- The **Read method** reads only the next character from a file and returns the integer code for the character using the following general format:

ObjectVar.Read

  - ***ObjectVar*** is the name of a **StreamReader** object
  - Use the **Chr function** to convert the integer code to a character

```
Dim textFile As StreamReader
Dim strInput As String = String.Empty
textFile = File.OpenText("names.txt")
Do While textFile.Peek <> –1
   strInput &= Chr(textFile.Read)
Loop
textFile.Close()
```

Other **StreamReader** Methods

- The **ReadToEnd method** reads and returns the entire contents of a file beginning at the current read position using the following general format:

ObjectVar.ReadToEnd

  - ***ObjectVar*** is the name of a **StreamReader** object

```
Dim textFile As StreamReader
Dim strInput As String
textFile = File.OpenText("names.txt")
strInput = textFile.ReadToEnd()
textFile.Close()
```

Working with Arrays and Files

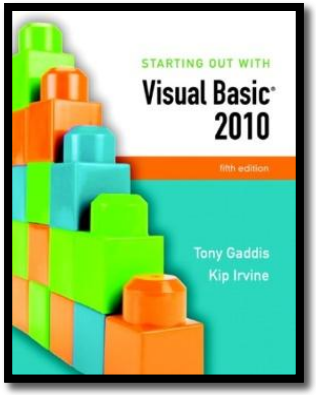
- The contents of an array can easily be written to a file using a loop

```
Dim outputFile as StreamWriter
outputFile = File.CreateText("Values.txt")

For intCount = 0 To (intValues.Length – 1)
   outputFile.WriteLine(intValues(intCount))
Next

outputFile.Close()
```

Working with Arrays and Files

- And it is just as easy to read the contents of a file into an array using a loop

```
Dim inputFile as StreamReader
inputFile = File.OpenText("Values.txt")

For intCount = 0 To (intValues.Length – 1)
   intValues(intCount) = CInt(inputFile.ReadLine())
Next

inputFile.Close()
```



Section 9.2

THE OPENFILEDIALOG, SAVEFILEDIALOG, FONTDIALOG, AND COLORDIALOG CONTROLS

Visual Basic provides dialog controls that equip your applications with standard Windows dialog boxes for operations such as opening files, saving files, and selecting fonts and colors.

Addison Wesley
is an imprint of

PEARSON

The OpenFileDialog and SaveFileDialog Controls

- Windows has a standard method of allowing a user to choose a file to open or save
- These methods let users browse for a file
  - The **OpenFileDialog control** and **SaveFileDialog control** provide this capability in Visual Basic
- To use the OpenFileDialog control
  - Double click on the *OpenFileDialog* tool in the *Toolbox* under the *Dialogs* tab
  - Appears in component tray
  - Use **ofd** as standard prefix when naming
- SaveFileDialog is used in a similar way

Displaying an Open Dialog Box

- Display control with the **ShowDialog** method

  ControlName.ShowDialog()

- Method returns a value indicating which dialog box button the user selects:
  - **Windows.Forms.DialogResult.OK** for the *OK* button
  - **Windows.Forms.DialogResult.Cancel** for the *Cancel* button
- For example:

```
If ofdOpenFile.ShowDialog() = Windows.Forms.DialogResult.OK Then
   MessageBox.Show(ofdOpenFile.FileName)
Else
   MessageBox.Show("You selected no file.")
End If
```

The Filter Property

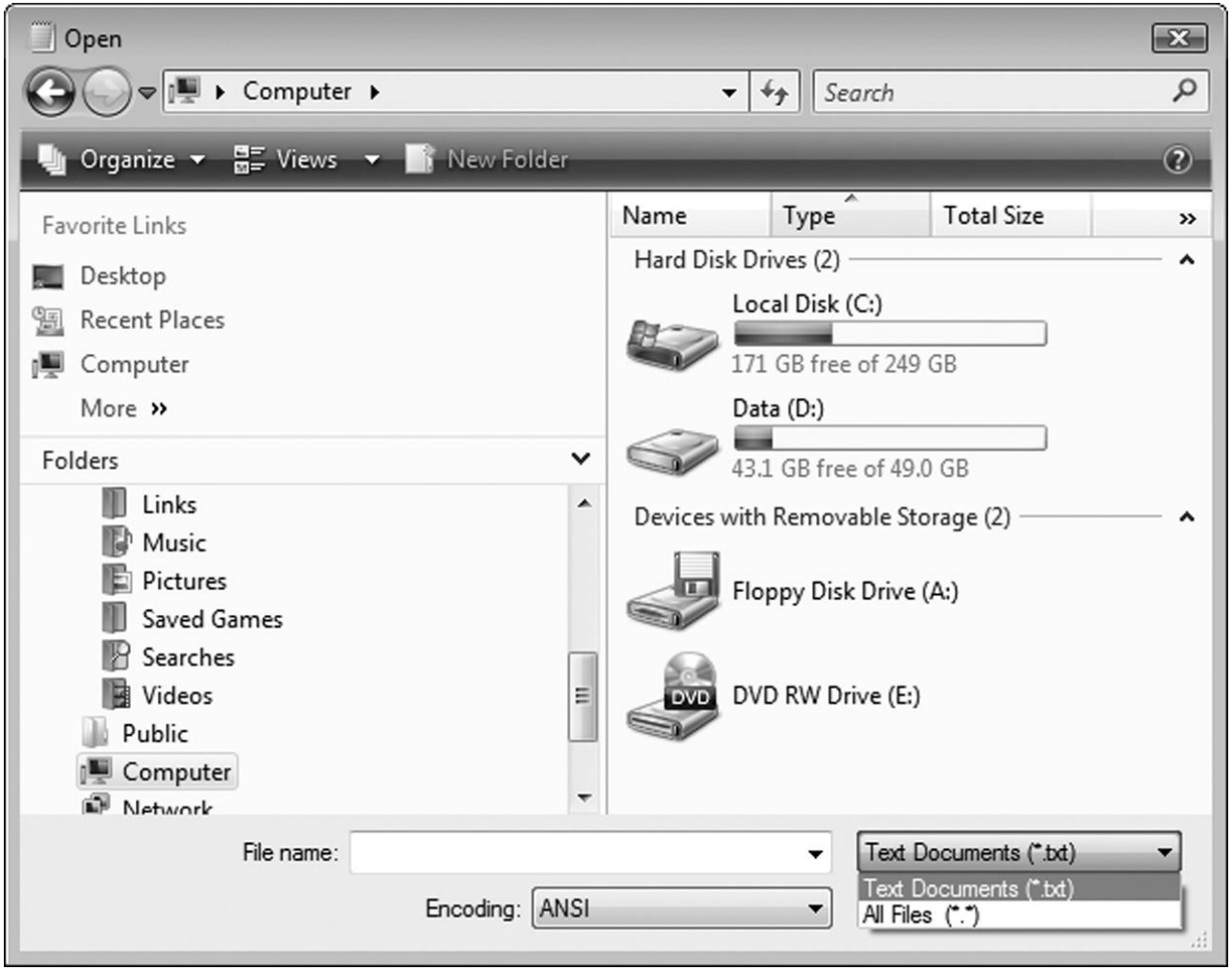
- FileDialog controls have a **Filter property**
  - Limits files shown to specific file extensions
  - Specify filter description shown to user first
  - Then specify the filter itself
  - Pipe symbol (|) used as a delimiter
- Following Filter property lets user choose:
  - *Text files (*.txt),* displays all .txt files
  - *All files (*.*),* displays all file extensions

Other OpenFileDialog Properties

- The **InitialDirectory property** is the initially displayed folder
- The **Title property** specifies the text on the title bar
  - The following example sets the Filter, InitialDirectory and Title properties:

```
' Configure the Open dialog box and display it.
With ofdOpenFile
   .Filter = "Text files (*.txt)|*.txt|All files (*.*)|*.*"
   .InitialDirectory = "C:\Data"
   .Title = "Select a File to Open"
   If.ShowDialog() = Windows.Forms.DialogResult.OK Then
      inputFile = File.OpenText(.Filename)
   End If
End With
```

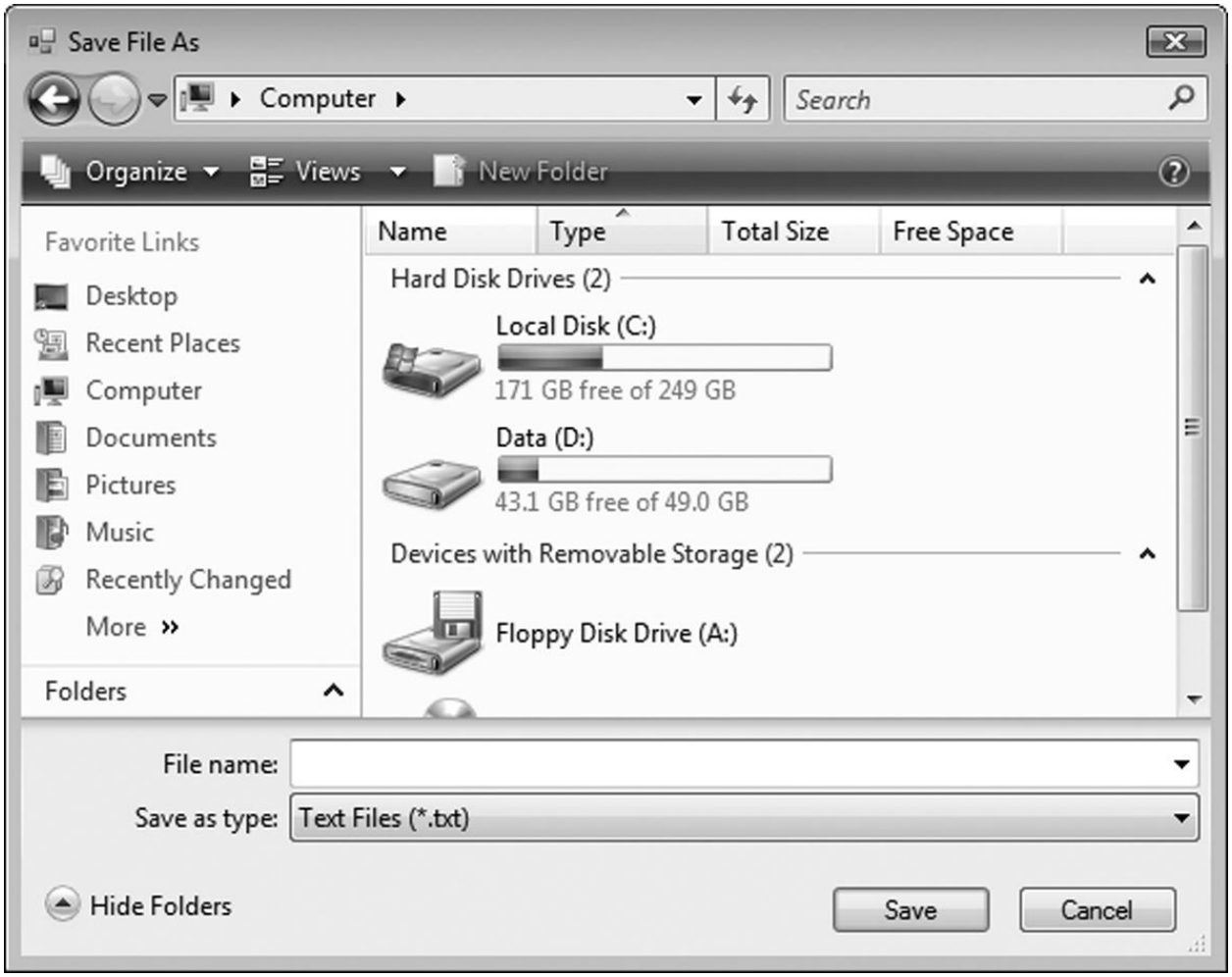
Open Dialog Box Example



The SaveFileDialog Control

- The *SaveFileDialog* uses the same methods:
  - **ShowDialog**
- The same properties:
  - Filter
  - InitialDirectory
  - Title
- And the same result constants:
  - **Windows.Forms.DialogResult.OK**
  - **Windows.Forms.DialogResult.Cancel**
- Tutorial 9-4 uses these controls in a text editor

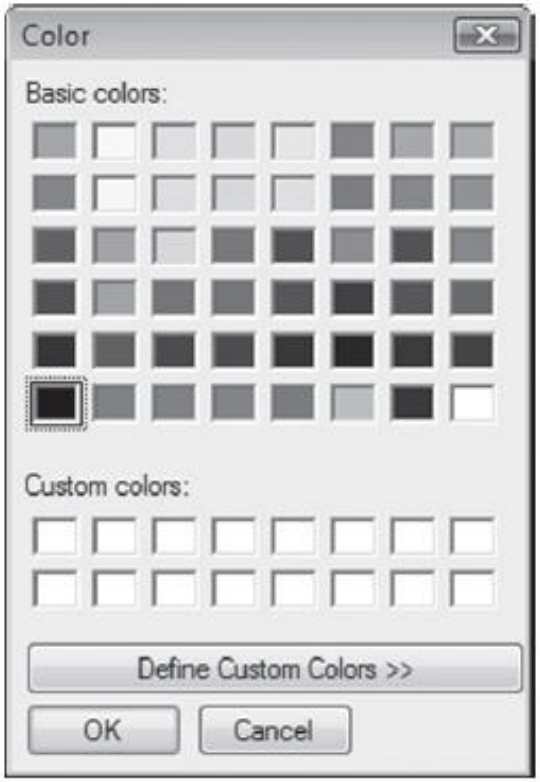
Windows *Save As* Dialog Box Example



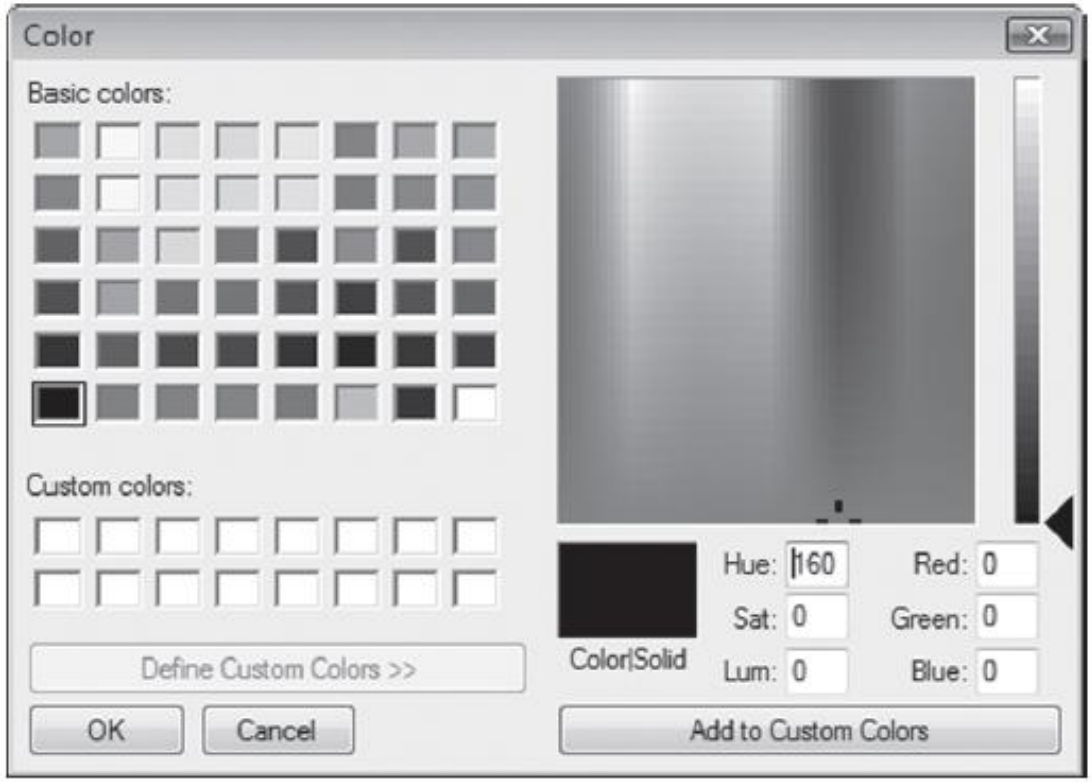
The ColorDialog Control

- The **ColorDialog control** displays a standard Windows ***Color Dialog box***
  - To place a ColorDialog control on a form
    - Double-click the *ColorDialog* icon in the *Dialogs* section of the *Toolbox*
    - Control appears in the component tray
    - Use the prefix **cd** when naming the control
  - To display a Color dialog box, call the **ShowDialog** method
    - Returns one of the following values
      - **Windows.Forms.DialogResult.OK**
      - **Windows.Forms.DiallogResult.Cancel**

Windows *Color* Dialog Box Example



Color dialog box

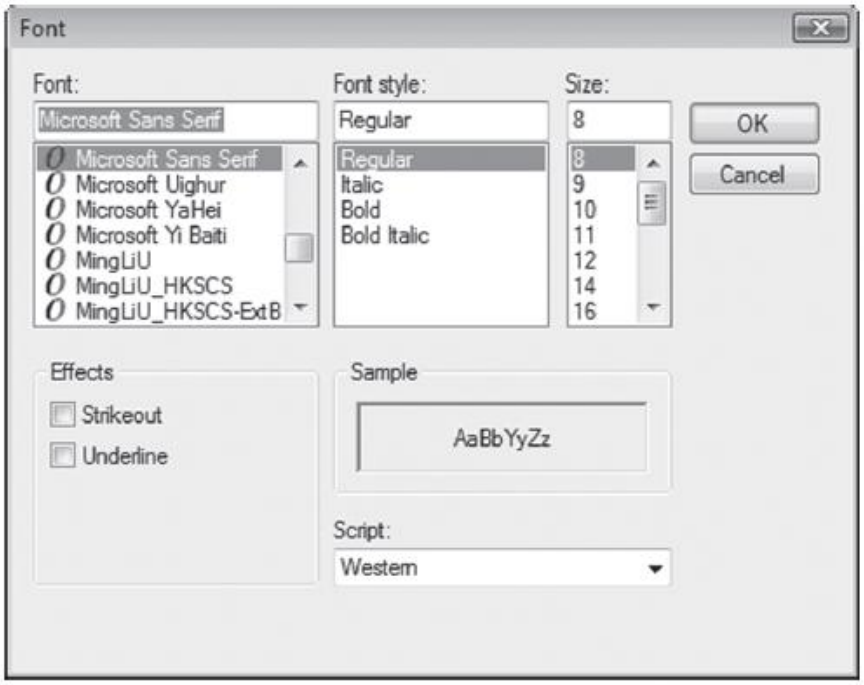


Fully open *Color* dialog box

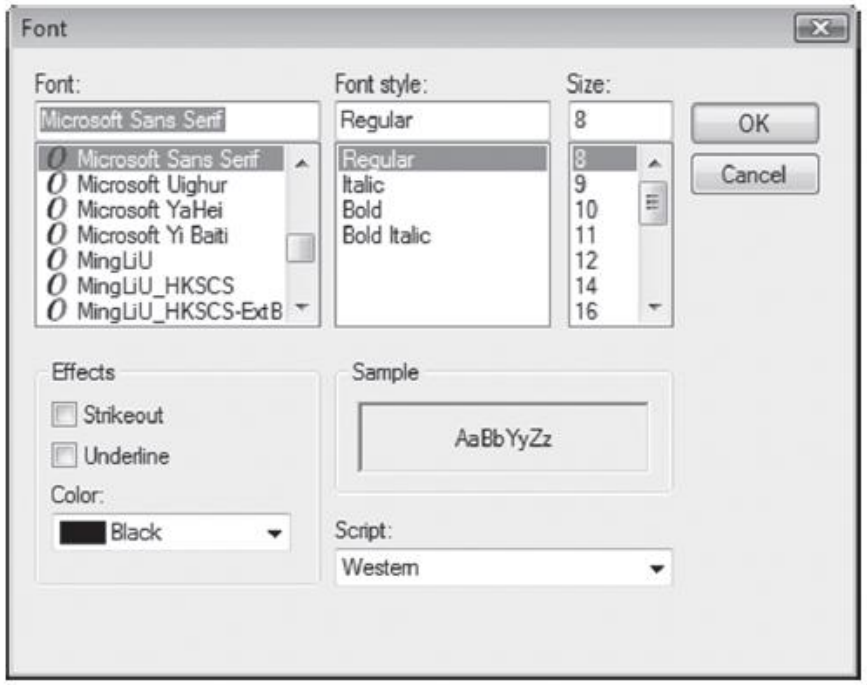
The FontDialog Control

- The **FontDialog control** displays a standard Windows ***Font* Dialog box**
  - To place a FontDialog control on a form
    - Double-click the *FontDialog* icon in the *Dialogs* section of the *Toolbox*
    - Control appears in the component tray
    - Use the prefix **fd** when naming the control
  - To display a Color dialog box, call the **ShowDialog** method
    - Returns one of the following values
      - **Windows.Forms.DialogResult.OK**
      - **Windows.Forms.DiallogResult.Cancel**

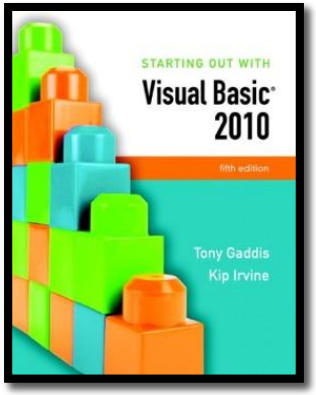
Windows *Font* Dialog Box Example



Default *Font* dialog box



Font dialog box with color choices displayed



Section 9.3

THE PRINTDOCUMENT CONTROL

The PrintDocument control allows you to send output to the printer.

Addison Wesley
is an imprint of

PEARSON

The PrintDocument Control

- The **PrintDocument control** gives your application the ability to print output on the printer
  - To place a PrintDocument control on a form
    - Double-click the *PrintDocument* tool in the *Printing* section of the *Toolbox*
    - Appears in the component tray
    - Use the prefix **pd** when naming the control

The **Print** Method and the **PrintPage** Event

- The PrintDocument control has a **Print method** that starts the printing process using the following general format:

  PrintDocumentControl.Print()

  - When the method is called, it triggers a PrintPage event
  - You must write code in the event handler to initiate printing
- To create a PrintPage event handler code template:
  - Double-click the PrintDocument control in the component tray
  - The event handler code template appears in the *Code* window:

```
Private Sub pdPrint_PrintPage(...) Handles pdPrint.PrintPage

End Sub
```

The **Print** Method and the **PrintPage** Event

- Inside the **PrintPage event hander**
  - You write code that sends text to the printer
    - Using a specified
      - Font
      - Color
      - Location
  - With the **e.Graphics.DrawString** method

The **Print** Method and the **PrintPage** Event

- The **e.Graphics.DrawString** method uses the following general format:

  e.Graphics.DrawString(*String, New Font(FontName, Size, Style),*
  Brushes.Black, *HPos, VPos)*

  - ***String*** is the string to be printed
  - ***FontName*** is a string holding the name of the font to use
  - ***Size*** is the size of the font in points
  - ***Style*** is the font style (bold, italic, regular, strikeout, or underline)
  - **Brushes.Black** specifies that the output should be printed in black
  - ***Hpos*** is the horizontal position of the output, in points, from the left margin
  - ***Vpos*** is the vertical position of the output, in points, from the top margin

- In Tutorial 9-5, you will modify the *Simple Text Editor* application from Tutorial 9-4 by adding a *Print* command to the *File* menu

PrintPage Event Handler Example

```
Dim inputFile As StreamReader          ' Object variable
Dim intX As Integer = 10               ' X coordinate for printing
Dim intY As Integer = 10               ' Y coordinate for printing

' Open the file.
inputFile = File.OpenText(strFilename)

' Read all the lines in the file.
Do While inputFile.Peek <> -1
   ' Print a line from the file.
   e.Graphics.DrawString(inputFile.ReadLine(),
                         New Font ("Courier", 10, FontStyle.Regular),
                         Brushes.Black, intX, intY)
   ' Add 12 to intY
   intY += 12
Loop

' Close the file.
inputFile.Close()
```

Formatted Reports with **String.Format**

- Reports typically contain the following sections:
  - A **report header**
    - Printed first, contains general information such as
      - The name of the report
      - The date and time the report was printed
  - The **report body**
    - Contains the report's data
      - Often formatted in columns
  - An optional **report footer**
    - Contains the sum of one for more columns of data

Printing Reports with Columnar Data

- Report data is typically printed in column format
- With each column having an appropriate header
- You can use Monospaced fonts to ensure that
  - Each character takes same amount of space
  - Columns will be aligned
- **String.Format** method is used to align data along column boundaries

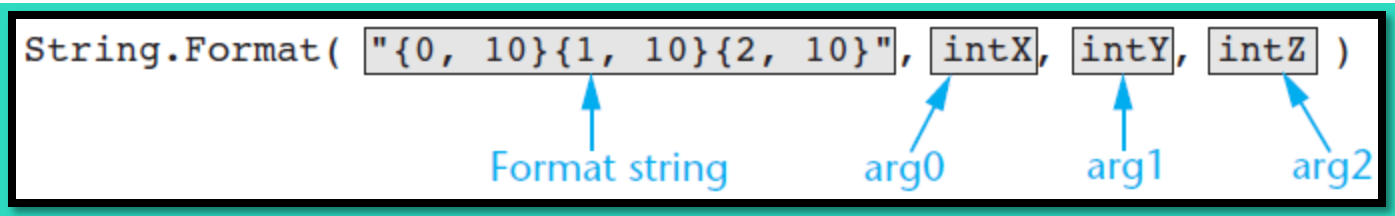
Using **String.Format** to Align Data along Column Boundries

- The **String.Format** method can be used to align data along column boundaries using the following general format:

String.Format(FormatString, Arg0, Arg1 [,...])

  - ***FormatString*** is a string containing the formatting specifications
  - ***Arg0*** and ***Arg1*** are values to be formatted
  - The ***[,…]*** notation indicates that more arguments may follow
  - The method returns a string that contains the data
  - Provided by the arguments (***Arg0***, ***Arg1***, etc)
  - Formatted with the specifications found in ***FormatString***

The Format String



- Contains three sets of numbers inside curly braces
  - The first number in a set specifies the argument index number
    - **0** represents the index for **intX**
    - **1** represents the index for **intY**
    - **2** represents the index for **intZ**
  - The second number in a set is an absolute value that specifies the column width, in spaces, and the type of justification that will be used
    - A positive number specifies right justification
    - A negative number specifies left justification

Example Report Header and Column Headings

```
Dim intCount As Integer                ' Loop counter
Dim decTotal As Decimal = 0            ' Accumulator
Dim intVertPosition As Integer         ' Vertical printing position

' Print the report header.
e.Graphics.DrawString("Sales Report",
                      New Font("Courier New", 12,FontStyle.Bold),
                      Brushes.Black, 150, 10)

e.Graphics.DrawString("Date and Time: " & Now.ToString(),
                      New Font("Courier New", 12, FontStyle.Bold),
                      Brushes.Black, 10, 38)

' Print the column headings.
e.Graphics.DrawString(String.Format("{0, 20} {1, 20} ","NAME", "SALES"),
                      New Font("Courier New", 12, FontStyle.Bold),
                      Brushes.Black, 10, 66)
```

Example Report Body and Footer

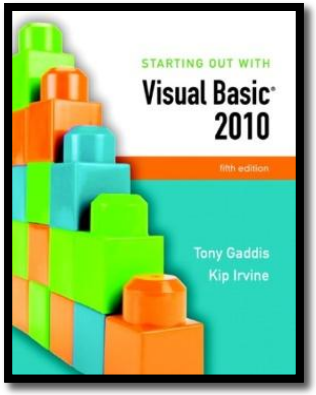
```
' Print the body of the report.
intVertPosition = 82

For intCount = 0 To 4
   e.Graphics.DrawString(String.Format("{0, 20} {1, 20}
                                ",strNames(intCount),decSales(intCount).ToString("c")),
                                New Font("Courier New", 12, FontStyle.Regular),
                                Brushes.Black, 10, intVertPosition)

   decTotal += decSales(intCount)
   intVertPosition += 14
Next

' Print the report footer.
e.Graphics.DrawString("Total Sales: " & decTotal.ToString("c"),
                              New Font("Courier New", 12, FontStyle.Bold),
                              Brushes.Black, 150, 165)
```

Example Report Output

Sales Report

Date and Time: 10/14/2010 11:12:34 AM

Name	Sales
John Smith	$2,500.00
Jill McKenzie	$3,400.00
Karen Suttles	$4,200.00
Jason Mabry	$2,200.00
Susan Parsons	$3,100.00
Total Sales:	$15,400.00



Section 9.4

STRUCTURES

Visual Basic allows you to create your own data types, into which you may group multiple data fields.

Addison Wesley
is an imprint of
PEARSON

Arrays vs. Structures

- Arrays:
  - Multiple fields in one array
  - All of the same data type
  - Distinguished by a numerical index
- Structures
  - Multiple fields in one structure
  - Can be of differing data types
  - Distinguished by a field name

Creating a Structure

- A **structure** is a data type you can create that contains one or more variables known as fields
- You create a structure at the class or module-level with the **structure statement**:

```
[AccessSpecifier] Structure StructureName
   FieldDeclarations
End Structure
```

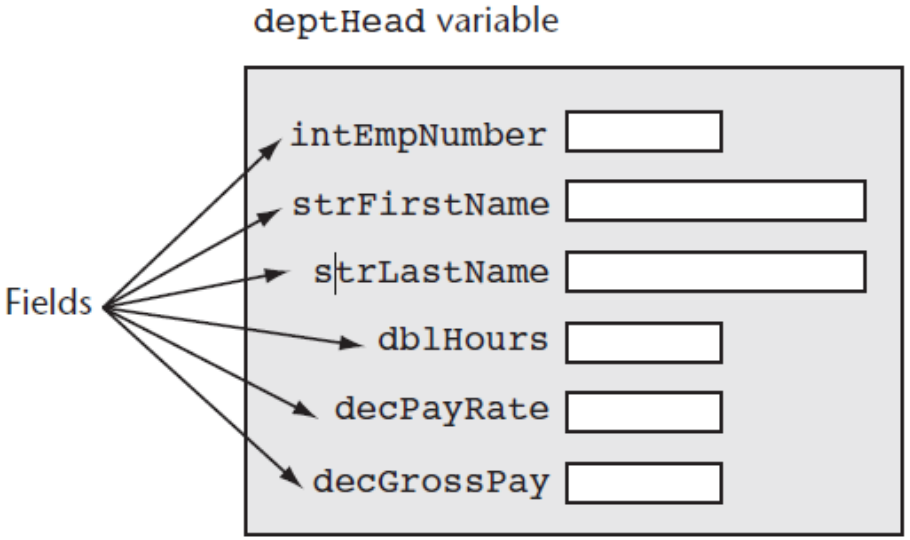
  - For example:

```
Structure EmpPayData
   Dim intEmpNumber As Integer
   Dim strFirstName As String
   Dim strLastName As String
   Dim dblHours As Double
   Dim decPayRate As Decimal
   Dim decGrossPay As Decimal
End Structure
```

Declaring a Structure

Dim deptHead As EmpPayData

Access each field with the dot operator



deptHead.intEmpNumber = 1101
deptHead.strFirstName = "Joanne"
deptHead.strLastName = "Smith"
deptHead.dblHours = 40.0
deptHead.decPayRate = 25
deptHead.decGrossPay = CDec(deptHead.dblHours) * deptHead.decPayRate

Passing Structure Variables to Procedures and Functions

- Structures can be passed to procedures and functions like any other variable
- The data type to use in the specification is the name of the structure

```
Sub CalcPay(ByRef employee As EmpPayData)
   ' This procedure accepts an EmpPayData variable
   ' as its argument. The employee's gross pay
   ' is calculated and stored in the grossPay
   ' field.
   With employee
      .decGrossPay =.dblHours * .decPayRate
   End With
End Sub
```

Arrays as Structure Members

- Structures can contain arrays
- Must **ReDim** after declaring structure variable

```
Structure StudentRecord
    Dim strName As String
    Dim dblTestScores() As Double
End Structure

Dim student As StudentRecord
ReDim student.dblTestScores(4)
student.strName = "Mary McBride"
student.dblTestScores(0) = 89.0
student.dblTestScores(1) = 92.0
student.dblTestScores(2) = 84.0
student.dblTestScores(3) = 96.0
student.dblTestScores(4) = 91.0
```

Arrays of Structures

- Can declare an array of structures
- Example below declares **employees** as an array of type **EmpPayData** with 10 elements

```
Const intMAX_SUBSCRIPT As Integer = 9
Dim employees(intMAX_SUBSCRIPT) As EmpPayData
```

  - To access individual elements in the array, use a subscript

```
employees(0).intEmpNumber = 1101
```

  - Us the **ReDim** statement to set the size of each array field

```
For intIndex = 0 To intMax_SUBSCRIPT
   ReDim students(intIndex).dblTestScores(4)
Next
```

- Tutorial 9-6 examines an application that uses a structure