CSC 330 Object-Oriented Design

C# Month Calendar
C# DateTimePicker Control
Data Files
The Calendar Controls - 1

- DateTimePicker and MonthCalendar controls display calendars on a form
  - DateTimePicker takes less screen space
    - Displays only day and date unless user drops down the calendar
  - Value property contains date
    - Control initially displays current date
    - User selects a date and the program retrieves the Value property
    - Assign a Date value to the property
MonthCalendar Control

- The MonthCalendar control (Fig. 1) displays a monthly calendar on the Form.
- Multiple dates can be selected by clicking dates on the calendar while holding down the Shift key.
### MonthCalendar properties and an event

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FirstDayOfWeek</strong></td>
<td>Sets which day of the week is the first displayed for each week in the calendar.</td>
</tr>
<tr>
<td><strong>MaxDate</strong></td>
<td>The last date that can be selected.</td>
</tr>
<tr>
<td><strong>MaxSelectionCount</strong></td>
<td>The maximum number of dates that can be selected at once.</td>
</tr>
<tr>
<td><strong>MinDate</strong></td>
<td>The first date that can be selected.</td>
</tr>
<tr>
<td><strong>MonthlyBoldedDates</strong></td>
<td>An array of dates that will displayed in bold in the calendar.</td>
</tr>
<tr>
<td><strong>SelectionEnd</strong></td>
<td>The last of the dates selected by the user.</td>
</tr>
<tr>
<td><strong>SelectionRange</strong></td>
<td>The dates selected by the user.</td>
</tr>
<tr>
<td><strong>SelectionStart</strong></td>
<td>The first of the dates selected by the user.</td>
</tr>
</tbody>
</table>

**Common MonthCalendar Event**

- **DateChanged**: Generated when a date is selected in the calendar.

---

**Fig. 2.** MonthCalendar properties and an event.
The Calendar Controls - 2

Displays only the day and date unless the user drops down the calendar. Saves screen space.
DateTimePicker Control

- The DateTimePicker control displays a calendar when a down arrow is selected.
- The DateTimePicker can be used to retrieve date and time information from the user.
### DateTimePicker Control

#### Properties and an event

<table>
<thead>
<tr>
<th>DateTimePicker Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalendarForeColor</td>
<td>Sets the text color for the calendar.</td>
</tr>
<tr>
<td>CalendarMonth Background</td>
<td>Sets the calendar’s background color.</td>
</tr>
<tr>
<td>CustomFormat</td>
<td>Sets the custom format string for the user’s options.</td>
</tr>
<tr>
<td>Date</td>
<td>The date.</td>
</tr>
<tr>
<td>Format</td>
<td>Sets the format of the date and/or time used for the user’s options.</td>
</tr>
<tr>
<td>MaxDate</td>
<td>The maximum date and time that can be selected.</td>
</tr>
</tbody>
</table>

**Fig. 3.** DateTimePicker properties and an event. (Part 1 of 2.)
### DateTimePicker properties and an event

<table>
<thead>
<tr>
<th>DateTimePicker properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MinDate</td>
<td>The minimum date and time that can be selected.</td>
</tr>
<tr>
<td>ShowCheckBox</td>
<td>Indicates if a CheckBox should be displayed to the left.</td>
</tr>
<tr>
<td>ShowUpDown</td>
<td>Indicates whether the control displays up and down buttons.</td>
</tr>
<tr>
<td>TimeOfDay</td>
<td>The time.</td>
</tr>
<tr>
<td>Value</td>
<td>The data selected by the user.</td>
</tr>
</tbody>
</table>

**Common DateTimePicker Event**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValueValueChanged</td>
<td>Generated when the Value property changes.</td>
</tr>
</tbody>
</table>

**Fig. 4.** DateTimePicker properties and an event. (Part 2 of 2.)
• Fig. 5 demonstrates using the DateTimePicker control to select an item’s drop-off time.

• The DateTimePicker has its Format property set to Long.

• In this application, the arrival date is always two days after drop-off, or three days if a Sunday is reached.
using System;
using System.Windows.Forms;

namespace DateTimePickerTest
{
    // constructor
    public partial class DateTimePickerForm : Form
    {
        InitializeComponent();
    }
} // end constructor
private void dateTimePickerDropOff_ValueChanged(object sender, EventArgs e)
{
    DateTime dropOffDate = dateTimePickerDropOff.Value;
    // add extra time when items are dropped off around Sunday
    if (dropOffDate.DayOfWeek == DayOfWeek.Friday || dropOffDate.DayOfWeek == DayOfWeek.Saturday || dropOffDate.DayOfWeek == DayOfWeek.Sunday)
    {
        // estimate three days for delivery
        outputLabel.Text = dropOffDate.AddDays(3).ToLongDateString();
    }
    else
    {
        // add extra time when items are dropped off around Sunday
        if (dropOffDate.DayOfWeek == DayOfWeek.Friday || dropOffDate.DayOfWeek == DayOfWeek.Saturday || dropOffDate.DayOfWeek == DayOfWeek.Sunday)
        {
            // estimate three days for delivery
            outputLabel.Text = dropOffDate.AddDays(3).ToLongDateString();
        }
        else
        {
            // estimate two days for delivery
            outputLabel.Text = dropOffDate.AddDays(2).ToLongDateString();
        }
    }
}

Fig. 5. Demonstrating DateTimePicker. (Part 2 of 4.)
private void DateTimePickerForm_Load(object sender, EventArgs e)
{
    // user cannot select days before today
    dateTimePickerDropOff.MinDate = DateTime.Today;

    // user can only select days of this year
    dateTimePickerDropOff.MaxDate = DateTime.Today.AddYears(1);

} // end method DateTimePickerForm_Load

class DateTimePickerForm
{
    // otherwise estimate only two days for delivery
    outputLabel.Text =
        dropOffDate.AddDays(2).ToLongDateString();

} // end method dateTimePickerDropOff_ValueChanged

Outline

Fig. 5. Demonstrating DateTimePicker. (Part 3 of 4.)

Setting the MinDate and MaxDate properties to keep drop-off sometime in the next year.
a) Clicking the down arrow
b) Selecting a day from the calendar
c) The Label updates
d) Selecting another day

Fig. 5. Demonstrating `DateTimePicker`. (Part 4 of 4.)
File I/O

- The user enters data into text boxes and the program stores the data in a file
  - writing or output
- The program retrieves the data from the file
  - reading or input
Simple File I/O

• .NET Framework includes classes to support simplified data file input and output (I/O)
• Use class methods to quickly and easily write and read data files
• `WriteAllText`, `WriteAllLines`, and `AppendAllText` are all methods of the `File` class
  – `WriteAllText` or `WriteAllLines` replaces any existing file
  – `AppendAllText` adds (appends) data to the end of a file

• Add using `System.IO` to the top of code or use `System.IO.File` for each reference
Writing and Reading Text

Files - 2

- Each method opens a file, creates a file (if one does not exist), writes the data, and closes the file
- Specify a complete path for the filename
  - If the path does not exist, an exception occurs
• To read data back into a program, use either `ReadAllText` or `ReadAllLines` method
  – `ReadAllText` reads entire file into a single string
  – `ReadAllLines` reads the file into a string array
  – Methods will fail if file or path does not exist
    • Place methods in a `try/catch` block
Viewing the Contents of a File

• By default, the new data file is created in the bin/Debug folder of the project
• View the data file by opening it from the Solution Explorer
  – File will appear in the Editor window
  – File can also be viewed by using a text editor, such as Notepad
.NET File Handling

- Simple file handling works well for small amounts of data
- Use streams for more robust and universal file handling
  - Same for all .NET languages
• .NET uses streams for file input and output
• A *stream* is designed to transfer a series of bytes from one location to another
• Streams are classes with methods and properties
  – Stream classes are found in the *System.IO* namespace
  – Add a *using* statement to the top of the file before the *namespace* statement

```csharp
using System.IO;
namespace Ch11WriteFile
```
File Handling Using Streams - 2

- **StreamWriter class**
  - Write StreamWriter code first, to create the data file

- **StreamReader class**
  - Write StreamReader code to read the file just created
Writing Data in a File Using a StreamWriter

- Declare a new StreamWriter object, also declares the name of the data file
- Use the StreamWriter’s `WriteLine` method to copy the data to a buffer
- Call the StreamWriter’s `Close` method
  - Transfers the data from the buffer to the file
  - Releases the system resources used by the stream
Instantiating a StreamWriter Object - 1

• The default location for the file is where the executable (.exe) is placed
  – The bin/Debug folder beneath the project folder
  – Can specify the complete path of the file

• General Form

```csharp
StreamWriter ObjectName = new StreamWriter("FileName");
StreamWriter ObjectName = new StreamWriter("FileName", BooleanAppend);
```

• To append data to an existing file, specify true for BooleanAppend
Instantiating a new `StreamWriter` object opens the file
- File must be open to write in it

- Instantiate the `StreamWriter` object *without* a path at the class level or inside a method
  - If the file does not exist, a new one is created in the project’s `bin\Debug` folder
    - No exception occurs whether or not the file exists

- Instantiate a `StreamWriter` object *with* a path in a method, inside a `try/catch` block
  - If the specified path does not exist, an exception is generated
Instantiating a StreamWriter Object - 3

• Examples

```
StreamWriter phoneStreamWriter = new StreamWriter("Phone.txt");
StreamWriter namesStreamWriter = new StreamWriter("C:\MyFiles\Names.txt");
StreamWriter logStreamWriter = new StreamWriter("C:\MyFiles\LogFile.txt", true);
StreamWriter namesStreamWriter = new StreamWriter(@"C:\MyFiles\Names.txt");
```

• To include a literal backslash in a string
  – Use the escape sequence (\\) or
  – Place an at-sign (@) before the literal or
  – Replace the backslash in the path with a forward slash (easiest solution)
• *Write* method places items consecutively in the file with no delimiter (separator)

• *WriteLine* method places an Enter between items
  – *WriteLine* method will be used in this chapter

• *DataToWrite* argument may be string or numeric
  – *WriteLine* method converts any numeric data to string and writes string data in the file
The **Save** button writes the data from the screen to the StreamWriter object and then clears the screen.

```csharp
private void saveButton_Click(object sender, System.EventArgs e)
{
    // Save the record to the file.
    phoneStreamWriter.WriteLine(nameTextBox.Text);
    phoneStreamWriter.WriteLine(phoneTextBox.Text);
    nameTextBox.Clear();
    phoneTextBox.Clear();
    nameTextBox.Focus();
}
```
Closing a File

- After finishing writing the data in a file, use the StreamWriter's `Close` method
  - Finishes writing all data from the stream's buffer to the disk and releases system resources
- Common location for the `Close` method is the program’s `Exit` command or the FormClosing event handler
- Failing to close the file may make the file unusable
- Best to open a file only when it is needed and close it as soon as writing to it is finished
  - Avoids tying up system resources unnecessarily
Reading Data from a File Using a StreamReader

- Declare an object of the StreamReader class
  - The constructor declares the filename and optional path
  - Opens the file
- Use the `ReadLine` method to read the data
  - May need a loop to retrieve multiple records
- Close the stream using the `Close` method
Declaring and Instantiating a StreamReader Object - 1

- The file must exist in the specified location or an exception occurs
- Instantiate the StreamReader object in a method enclosed in a `try/catch` block
- General Form

```csharp
//Declare a class-level variable.
Private StreamReader ObjectName;

//Inside a method.
ObjectName = new StreamReader("FileName");
```
• Examples

```csharp
try
{
    StreamReader namesStreamReader = new StreamReader("C:\MyFiles\Names.txt");
}
catch
{
    MessageBox.Show("File does not exist.");
}
// Declare a class-level variable.
StreamReader phoneStreamReader;
...
// In a method, to catch an exception for a missing file.
try
{
    phoneStreamReader = new StreamReader("Phone.txt");
}
catch
{
    MessageBox.Show("File does not exist");
}
```
Using the ReadLine Method

• Use the StreamReader's `ReadLine` method to read previously saved data
  – Each time it executes, it reads the next line from the file
• Assign the value from the read to the desired location, such as a label, text box, or string variable
• The `ReadLine` method has no arguments

```csharp
nameTextBox.Text = phoneStreamReader.ReadLine();
```
Checking for the End of the File

• Use the StreamReader's *Peek method*
  – Looks at the next element without reading it
  – If you peek beyond the last element, the value returned is negative 1 (-1)
    
    ```csharp
    if (phoneStreamReader.Peek() != -1)
    ```

• Read elements in the exact same order as they were written to obtain valid data

• The ReadLine method does not throw an exception when an attempt is made to read past the end of the file
The File Read Program

• The program reads the file and displays records on a form
• Each time the user clicks Read Next, the program reads and displays the next record
Using the File Common Dialog Box

- Allows the user to browse and enter the filename at run time
- Use the `OpenFileDialog` common dialog component to display the dialog box
  - Use the object's `FileName` property to open the selected file
# OpenFileDialog Component Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of the component. You can use the default openFileDialog1.</td>
</tr>
<tr>
<td>CheckFileExists</td>
<td>Display an error message if the file does not exist. Set to false for saving a file since you want to create a new file if the file does not exist. Leave at the default true to read an existing file.</td>
</tr>
<tr>
<td>CheckPathExists</td>
<td>Display an error message if the path does not exist. Set to false for saving a file since you want it to create the new folder if necessary.</td>
</tr>
<tr>
<td>FileName</td>
<td>The name of the file selected or entered by the user, which includes the file path. Use this property after displaying the dialog box to determine which file to open. You also can give this property an initial value, which places a default filename in the dialog box when it appears.</td>
</tr>
<tr>
<td>Filter</td>
<td>Filter file extensions to display. Example: Text Files (*.txt)</td>
</tr>
<tr>
<td>InitialDirectory</td>
<td>Directory to display when the dialog box opens. Set this in code to System.IO.Directory.GetCurrentDirectory() to begin in the current directory.</td>
</tr>
<tr>
<td>Title</td>
<td>Title bar of the dialog box.</td>
</tr>
</tbody>
</table>


Displaying the Open File Dialog Box - 1

• Add an OpenFileDialog component to the form
  – Appears in the form's component tray
• At design time, set initial properties for Name, CheckFileExists, CheckPathExists, Filter, and Title
• In code set the InitialDirectory property to `Directory.GetCurrentDirectory`
  – If the program does not include a `using` statement for System.IO, fully qualify the name
    • `System.IO.Directory.GetCurrentDirectory()`
Displaying the Open File Dialog Box - 2

- Display the dialog box using the `ShowDialog` method and retrieve the `FileName` property.
- A user may click on the `Cancel` button of the `Open File` dialog box.
  - Check the `DialogResult` for `Cancel`.
  - If the user clicks `Cancel`, there is one more task for the program.
    - A `StreamWriter` object that is not open cannot be closed.
Checking for Successful File Open

- If the file does not open, StreamWriter is not instantiated
- Verify the object’s instantiation using the C# keyword `null`
  - An object variable that is not instantiated has a value of `null`
- Place code to check for `null` in the form’s FormClosing event handler
Checking for an Already Open File

• If a user selects *File/Open* twice, it can cause a problem
  – A second open instantiates another file stream
  – The *Close* method never executes for the first file
• Check for an active instance of the file stream before instantiating a new one

```csharp
if (phoneStreamWriter != null) // Is the file already open?
{
    phoneStreamWriter.Close();
}
// Open the file stream here.
```
Using the SaveFileDialog Component

- Displays the standard system *Save File As* dialog box
- The SaveFileDialog allows the user to browse and enter a filename to save
- Has most of the same properties as the OpenFileDialog component
- By default, the SaveFileDialog component checks for an already-existing file and displays a dialog box asking the user whether to replace the existing file
Saving the Contents of a List Box

• Save the contents of a list when the program exits and reload the list when the program reopens
  – Do not assign values to the list's Items collection at design time
    • When the program begins, open the data file and read the list elements into the Items collection
  – If the file holding the list of elements does not exist, allow the user to create a new list by adding items
  – If the user makes any changes to the list, ask whether to save the list when the program ends
  – Include a menu item to save the list
Loading the List Box

• Read the file into the list in the Form_Load method
• Loop through the file until all elements are read
• Place each item in the list with the Items.Add method

```csharp
while (classesStreamReader.Peek() != -1)
{
    classesComboBox.Items.Add(classesStreamReader.ReadLine());
}
```
Checking for Existence of the File

- When the StreamReader object is created, the constructor makes sure the file exists
- If it does not exist, give the user options
  - Locate the file and try again
  - Exit the program
  - Begin with an empty list, add items, and create a new file

```
responseDialogResult = MessageBox.Show("Create a new file?", "File not Found",
MessageBoxButtons.YesNo,
MessageBoxIcon.Question);
```
Saving the File

• Provide a menu option for the user to save the file
• Open a StreamWriter object and loop through the Items collection of the list box
  – Save each element with a WriteLine method
Querying the User to Save

• When the program ends, check whether any changes were made
  – Use a class-level Boolean variable
  – Call the variable isDirtyBoolean with a value of false
    • In each method that allows change (Add, Remove, Clear) set isDirtyBoolean to true
    • After saving the file, set the variable to false
  – Just before the project ends, check the value of isDirtyBoolean
    • If true, give the user the option to save the file
    • If false, exit without a message
The Form_FormClosing Event Handler

- Best location to ask the user to save the file is in the form’s FormClosing event handler
  - Better than exit method because a user can quit the program in more than one way
- The FormClosing event handler executes when the user clicks on an Exit button or menu command, the window’s Close button, or even exits Windows
Windows Presentation Foundation (WPF) - 1

• WPF provides ability to create richer user interfaces for multiple platform development
• Windows Vista uses WPF to bring better multimedia to the operating system
• WPF is available in Visual Studio and Microsoft Expression Studio
• Microsoft Silverlight is a scaled-down version of WPF
  – Rich Web-based interface, works with all leading browsers and on multiple platforms
  – Able to integrate vector-based graphics, media, text, animation and overlays into the Web interface
Windows Presentation Foundation (WPF) - 2

- Web pages are created in two parts, the interface and the application code
  - Designer creates the interface
  - Developer does the programming
  - Expression blend facilitates creating the two parts

- In Visual Studio templates exist for a WPF application and for WPF Browser Applications
Windows Presentation Foundation (WPF) - 3

• WPF user interfaces use **XAML** *(Extensible Application Markup Language)*
  – Much more interactive than traditional HTML

• **XBAP** refers to a **XAML Browser Application**
  – Runs in an Internet browser

• Technology allows creation of hybrid applications
WPF Interoperability

• Allows use of WPF controls in a Windows Forms application
  – ElementHost available by default
    • A container that allows addition of other WPF controls to the Windows Form
    – Add additional available controls at run time, rather than design time
• To use WPF Interoperability, add the ElementHost control to a Windows Form
  – Add the WPF controls in code
  – Include a using statement for System.Windows.Controls
WPF Interoperability - 2

- Expander control allows part of a page to show or be hidden
  - User clicks to expand the control, value of the Content property displays
Writing a WPF Application

• IDE layout for a WPF application resembles ASP.NET layout
• Document window is split, shows XAML and the design
  – Collapse the XAML screen
• Use the grid container to help with flow layout and to place controls
  – Grid lines can be set to desired height and width
• Many controls have same function and feel as Windows Forms controls
  – Extra properties available with WPF controls
Introduction

• The graphics system in WPF (Windows Presentation Foundation) is designed to use your computer’s graphics hardware to reduce the load on the CPU and speed up graphics rendering.

• WPF graphics are measured in **resolution-independent pixels**, where one pixel typically represents 1/96 of an inch.

• WPF graphics are rendered on screen using a **vector-based** system, allowing graphic elements to be preserved across any rendering size.

• **Raster-based** systems have the precise pixels specified for each graphical element. Raster-based graphics tend to degrade in appearance as they are scaled larger.

• WPF also includes 3-D modeling and rendering capabilities.