Department of Economic Informatics and Cybernetics
Bucharest University of Economic Studies

Windows Applications Programming

Windows Forms

Liviu-Adrian Cotfas, PhD.

liviu.cotfas@ase.ro

Few words about me…

https://ro.linkedin.com/in/cotfasliviu

Administrative issues

- API reference:
  - https://docs.microsoft.com/en-us/dotnet/api/?view=netframework-4.8

- Windows Forms source code:
  - https://referencesource.microsoft.com/#System.Windows.Forms

Windows Forms

.NET Graphical User Interface

- Windows Forms (2001)
- Windows Presentation Foundation
- Universal Windows Platform

A USER INTERFACE IS LIKE A JOKE.
IF YOU HAVE TO EXPLAIN IT,
IT'S NOT THAT GOOD.

Startup and Shutdown

- Startup

```
[STAThread]
static void Main( )
{
        Application.Run(new Form1( ));
}
```

Demo

- Implement a calculator (similar to the SimpleCalculator sample)

Application Class

Docs: https://docs.microsoft.com/en-us/dotnet/api/system.windows.forms.application

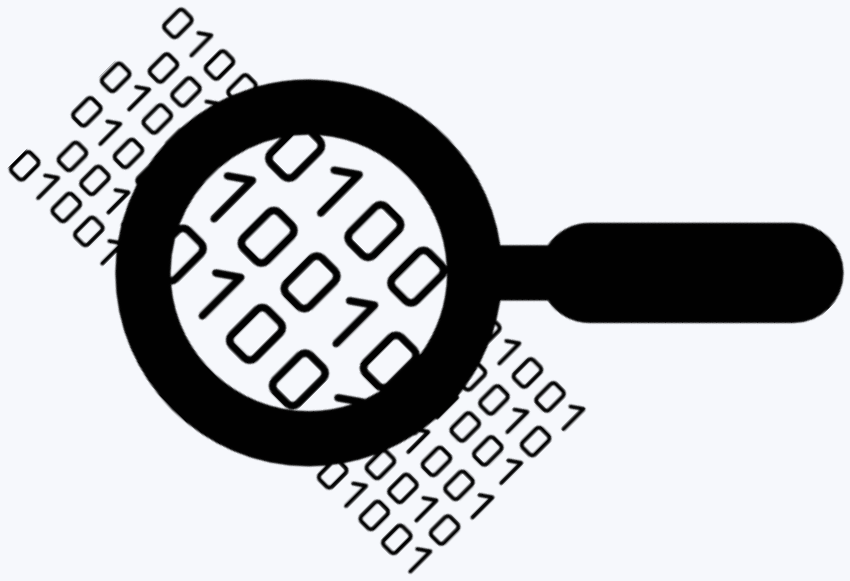
Important Methods:

- Run
- Exit

Important Events

- ApplicationExit
- ThreadException

Demo



- Check the difference between Form.Close() and Application.Exit()
- Subscribe to ApplicationExit
- Subscribe to ThreadException

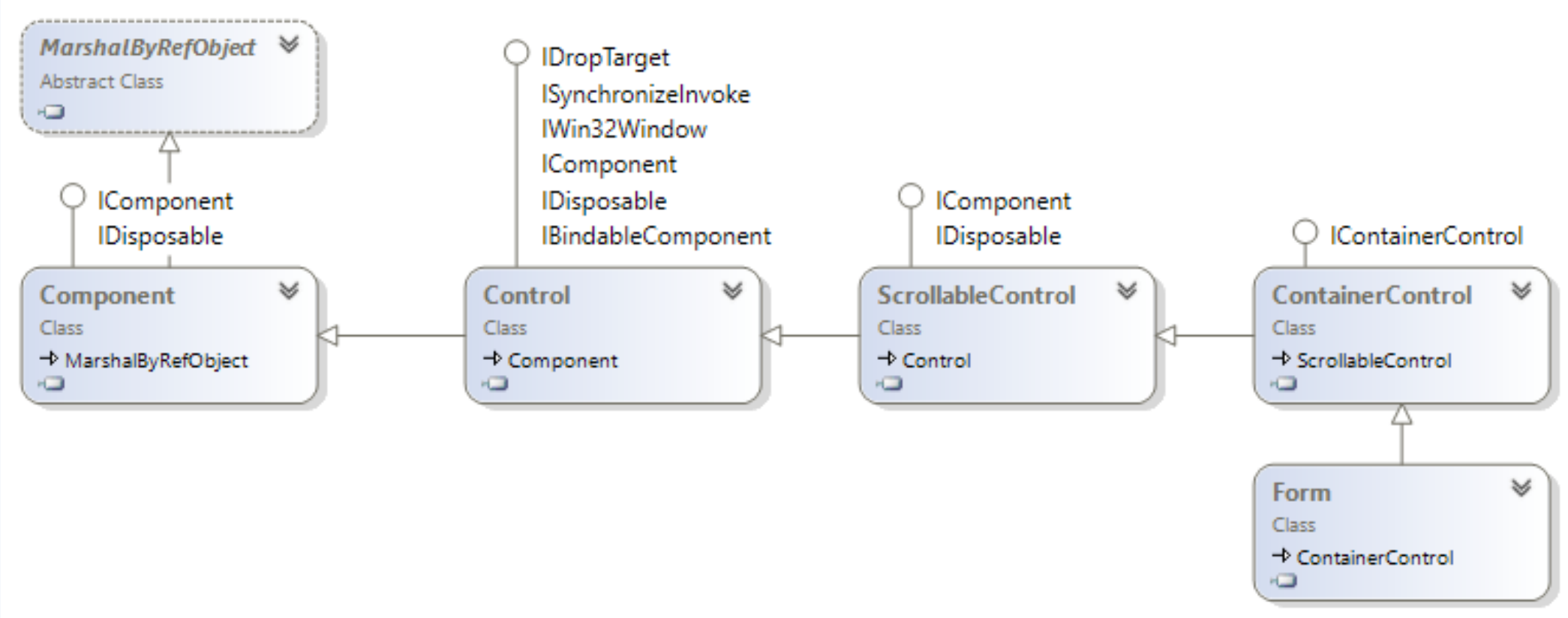
Form Class

Form Class

- All windows in a Windows Forms application are represented by objects of some type deriving from the Form class.

```
public partial class MainForm : Form
{
        // Constructor
        public MainForm()
        {
                InitializeComponent();
        }
}
```

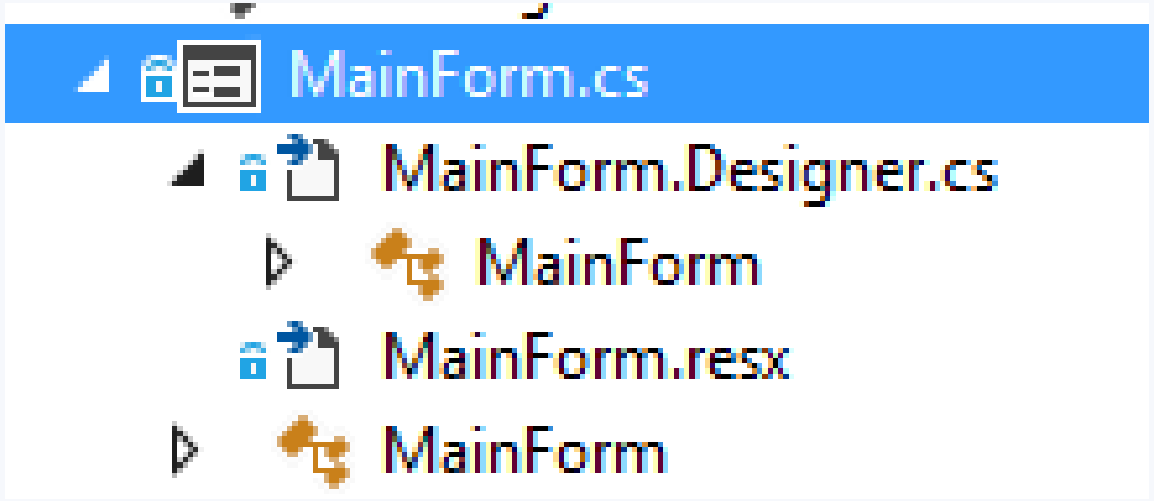
Inheritance

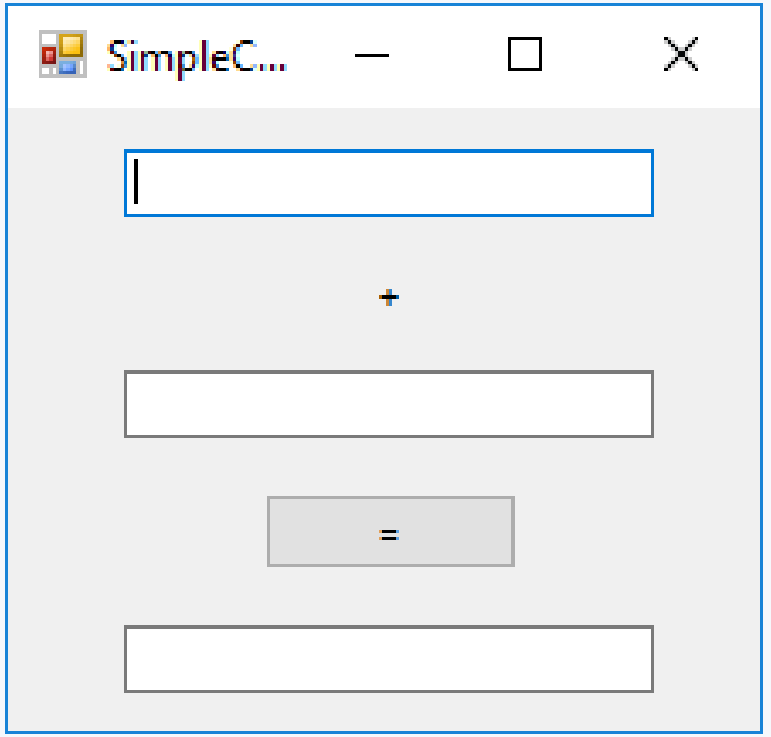


Further reading: link

The Forms Designer

- visual designer that auto-generates code.
- Uses **partial classes**.





Demo

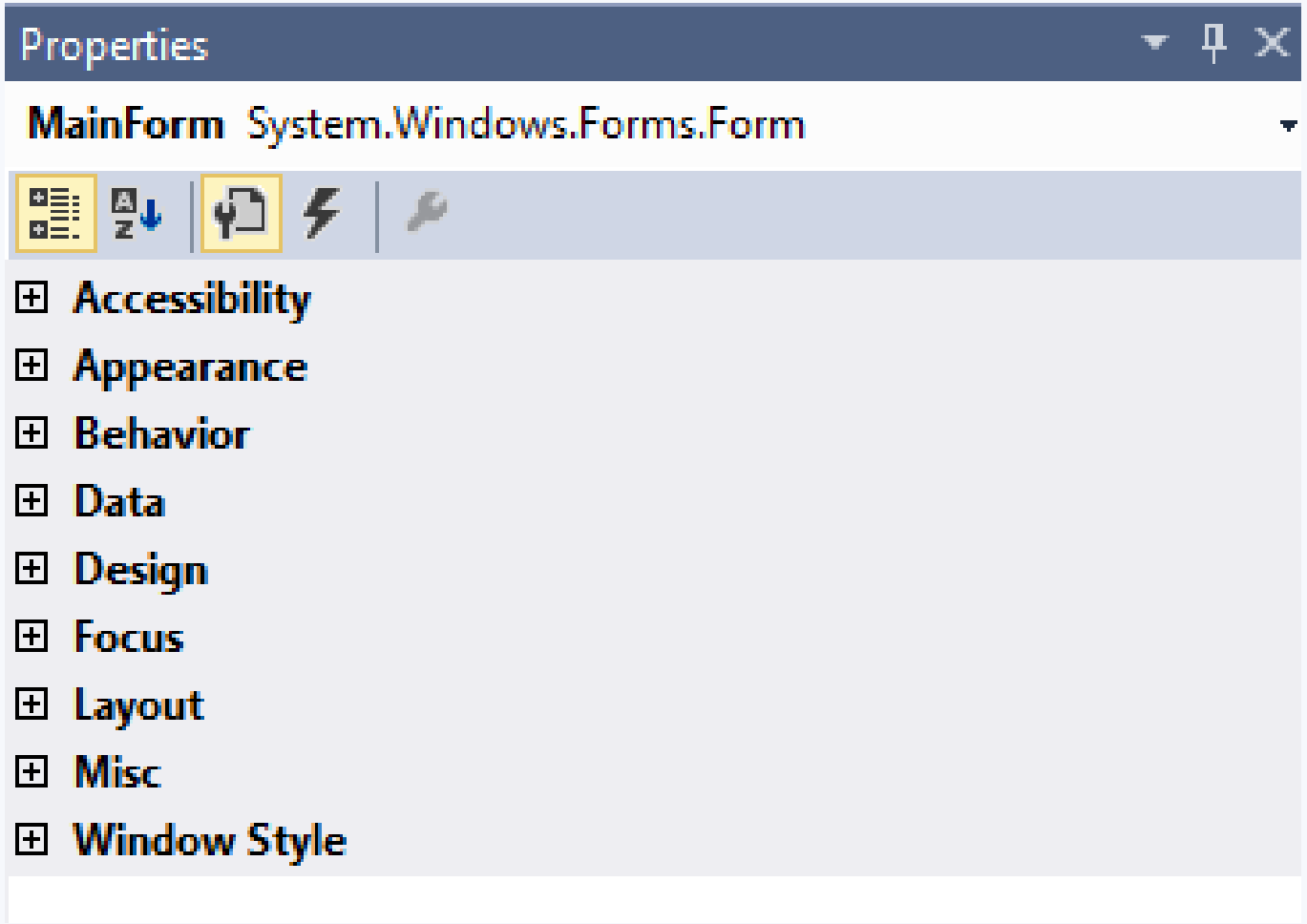
Partial Class

- It is possible to split the definition of a class or a struct, an interface or a method over two or more source files. Each source file contains a section of the type or method definition, and all parts are combined when the application is compiled.

```
public partial class Employee{
    public void DoWork()    {
    }
}

public partial class Employee{
    public void GoToLunch()    {
    }
}
```

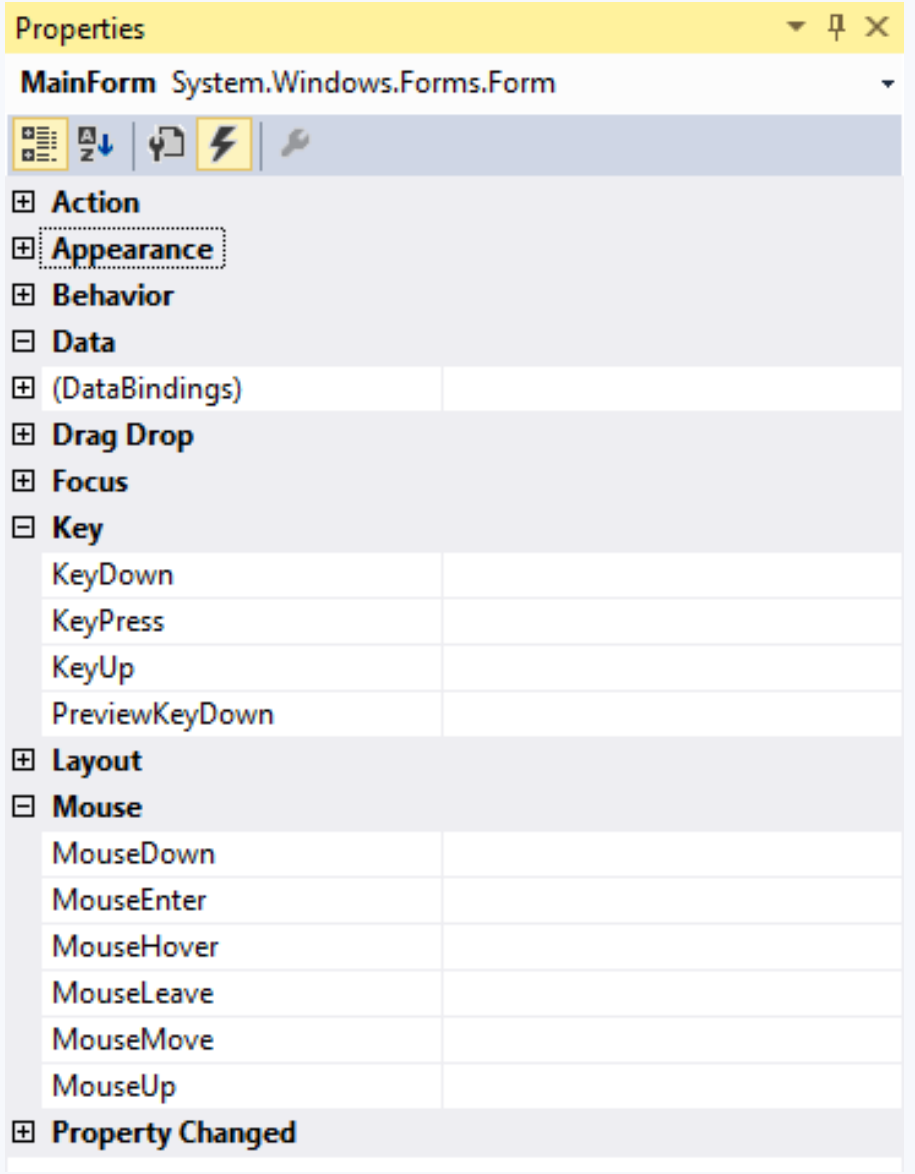
Properties

- Appearance
  - BackColor
  - Font
  - ForeColor
  - Text
- Design
  - Name



Events

- Load
- Key
  - KeyDown
  - KeyPress
  - KeyUp
- Mouse
  - MouseDown / MouseUp
  - MouseEnter / MouseLeave



Keyboard Events

Keyboard Events

- Key events occur in the order specified bellow.

Event	Event data	Description
KeyDown	KeyEventArgs	Raised when a key is pressed. The KeyDown event occurs prior to the KeyPress event.
KeyPress	KeyPressEventArgs	Raised when a character generating key is pressed. The KeyPress event occurs after the KeyDown event and before the KeyUp event.
KeyUp	KeyEventArgs	Raised when a key is released.

- **KeyDown** and **KeyPress** occur as long as a key is pressed.

Demo

- KeyEvents Sample

KeyPress

- Occurs when a character is pressed on the keyboard, and again each time the character is repeated while it continues to be pressed.
- Not raised by some non-character keys such as the left and right buttons.
- Use the KeyChar property to sample keystrokes at run time and to consume or modify a subset of common keystrokes.

KeyPressEventArgs

Property	Description
Handled	Boolean value indicating if the event was handled. false until set otherwise. When true, the keystroke is not displayed.
KeyChar	Read-only value of type char containing the composed ASCII character.

KeyDown and KeyUp

- **KeyDown** - Occurs when a key on the keyboard is pressed down.
- **KeyUp** - Occurs when a key on the keyboard is released.
- The **KeyDown** and **KeyUp** events are useful to fine-tune an application's behavior as keyboard keys are pressed and released, and for handling noncharacter keys such as the function or arrow keys.
- Handlers for these events receive an instance of the KeyEventArgs class as their event parameter.

KeyEventArgs properties

Property	Data type	Description
Alt	Boolean	Read-only value indicating if the Alt key was pressed. true if pressed, false otherwise.
Control	Boolean	Read-only value indicating if the Ctrl key was pressed. true if pressed, false otherwise.
Shift	Boolean	Read-only value indicating if the Shift key was pressed. true if pressed, false otherwise.
Modifiers	Keys	Read-only flags indicating the combination of modifier keys (Alt, Ctrl, Shift) pressed. Modifier keys can be combined using the bitwise OR operator.
Handled	Boolean	Value indicating if the event was handled. false until set otherwise.
KeyCode	Keys	Read-only value containing the key code for the key pressed. Typical values include the A key, Alt, and BACK (backspace).
KeyData	Keys	Read-only value containing the key code for the key pressed, combined with modifier flags to indicate combination of modifier keys (Alt, Ctrl, Shift).
KeyValue	integer	Key code property expressed as a read-only integer.

Demo

- Implement a simple NumericTextBox
- KeyEventsNumericTextBox Sample

Mouse Events

Mouse Events

Event	Event argument	Description
Click	EventArgs	Raised when the control is clicked.
DoubleClick	EventArgs	Raised when the control is double-clicked.
MouseEnter	EventArgs	Raised when the mouse cursor enters the control.
MouseHover	EventArgs	Raised when the mouse cursor hovers over the control.
MouseLeave	EventArgs	Raised when the mouse cursor leaves the control.
MouseDown	MouseEventArgs	Raised when the mouse cursor is over the control and a mouse button is pressed.
MouseMove	MouseEventArgs	Raised when the mouse cursor is moved over the control.
MouseWheel	MouseEventArgs	Raised when the control has focus and the mouse wheel is rotated.
MouseUp	MouseEventArgs	Raised when the mouse cursor is over the control and a mouse button is released.

Demo

- MouseEvents Sample
- Hint: use “Alt + R” to clear the list of events

Click and MouseClick

Depressing a mouse button when the cursor is over a control typically raises the following series of events from the control:

- MouseDown event.
- Click event.
- MouseClick event.
- MouseUp event.

Click events are logically higher-level events of a control. They are often raised by other actions, such as pressing the ENTER key when the control has focus.

MouseClick contains specific mouse information (button, number of clicks, wheel rotation, or location)

MouseEventArgs properties

Property	Description
Button	Returns the pressed mouse button. Must be one of the members of the MouseButtons enumeration
Clicks	Returns the integer number of times the mouse button was pressed and released. Resets after two clicks.
Delta	Returns the signed integer number of detents the mouse wheel was rotated. A positive value indicates that the wheel was rotated forward, i.e., away from the user, and a negative value indicates the wheel was rotated backward, i.e., toward the user.
X	The X coordinate, in pixels, of the mouse cursor's hot spot when the button was clicked, relative to the top-left corner of the control.
Y	The Y coordinate, in pixels, of the mouse cursor's hot spot when the button was clicked, relative to the top-left corner of the control.

Click

```
this.btnCalculate.Click += new System.EventHandler(this.btnCalculate_Click);
```

```
private void btnCalculate_Click(object sender, EventArgs e)
{
        var value1 = int.Parse(tbValue1.Text);
        var value2 = int.Parse(tbValue2.Text);

        var sum = value1 + value2;
        tbSum.Text = sum.ToString();
}
```

Alt Shortcuts

Alt Shortcuts

- An **access key** is an underlined character in the text of a menu, menu item, or the label of a control such as a button. It allows the user to "click" a button by pressing the ALT key in combination with the predefined access key.

```
// C#
// Set the letter "P" as an access key.
button1.Text = "&Print";
```

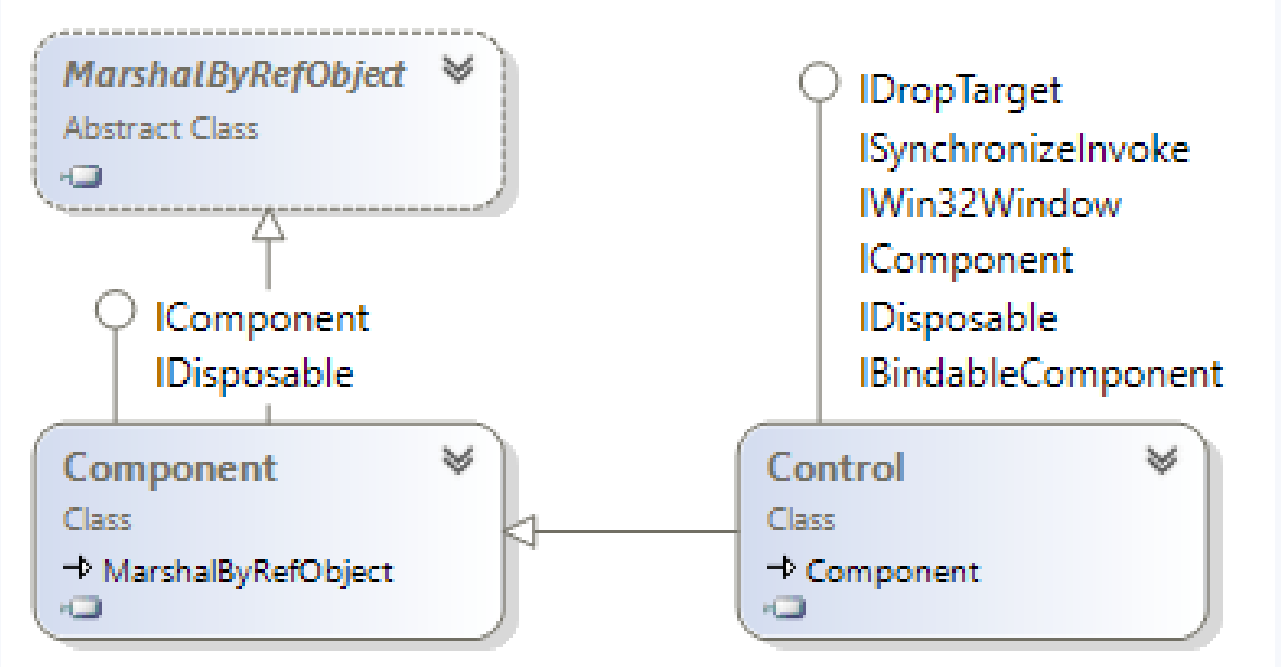
Demo

- In the SimpleCalculator sample replace the Text on the button with “&Calculate”

Controls

Control Class

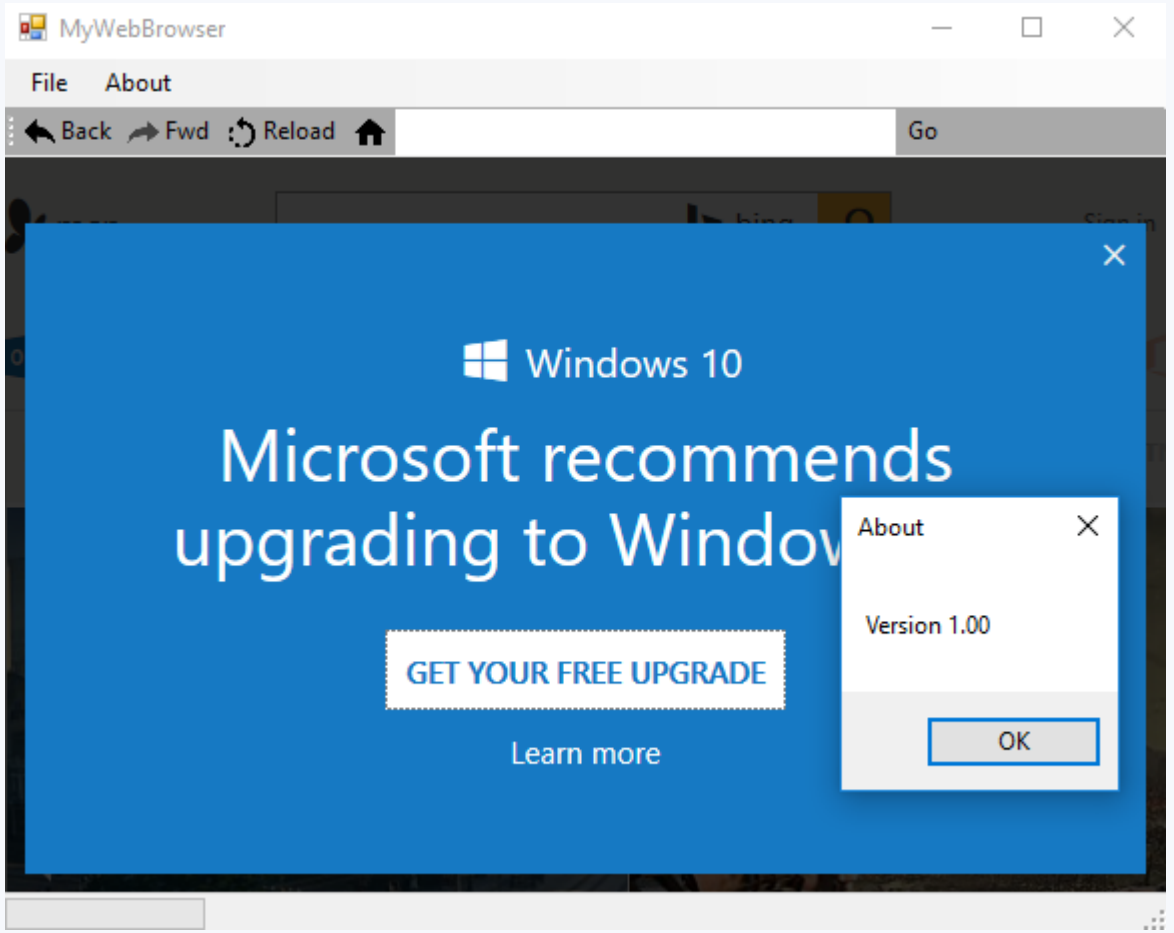
- The Control class forms the basis for all windows controls in .NET, and provides many properties, methods, and events.



- Further reading: link

Common controls

- Recommended Controls and Components by Function: docs.microsoft.com/en-us/dotnet/desktop/winforms/controls/windows-forms-controls-by-function



Common controls

- Alphabetic list of controls and components: docs.microsoft.com/en-us/dotnet/desktop/winforms/controls/controls-to-use-on-windows-forms

UI Design

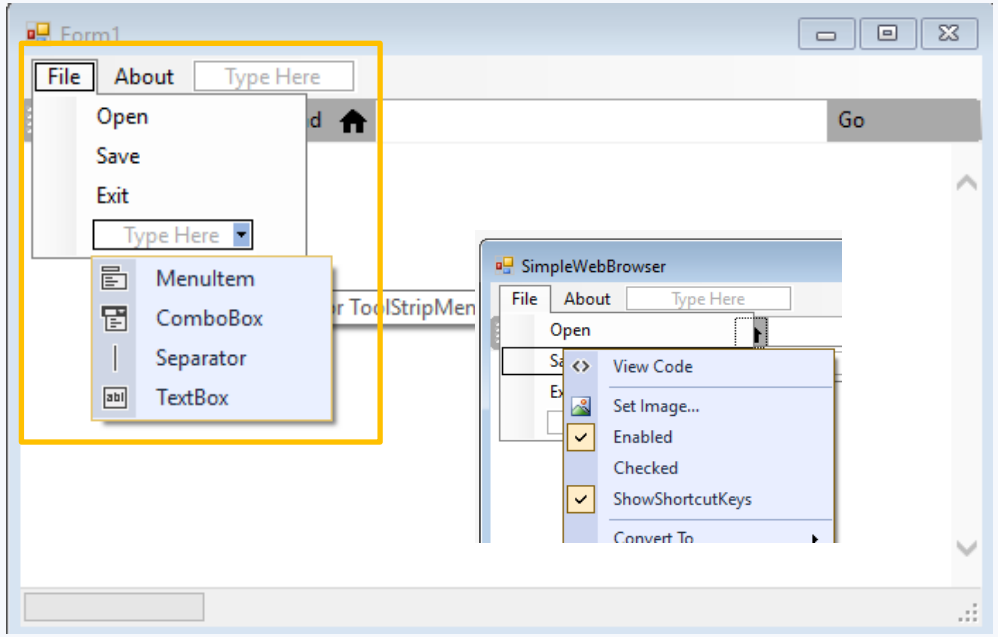
Alignment

Menus

Menus

- MenuStrip
- ToolStrip / ToolStripContainer
- ContextMenuStrip
- StatusStrip

MenuStrip



ToDo: Check how the types of MenuStrip items are used in Visual Studio and in Notepad.

MenuStrip

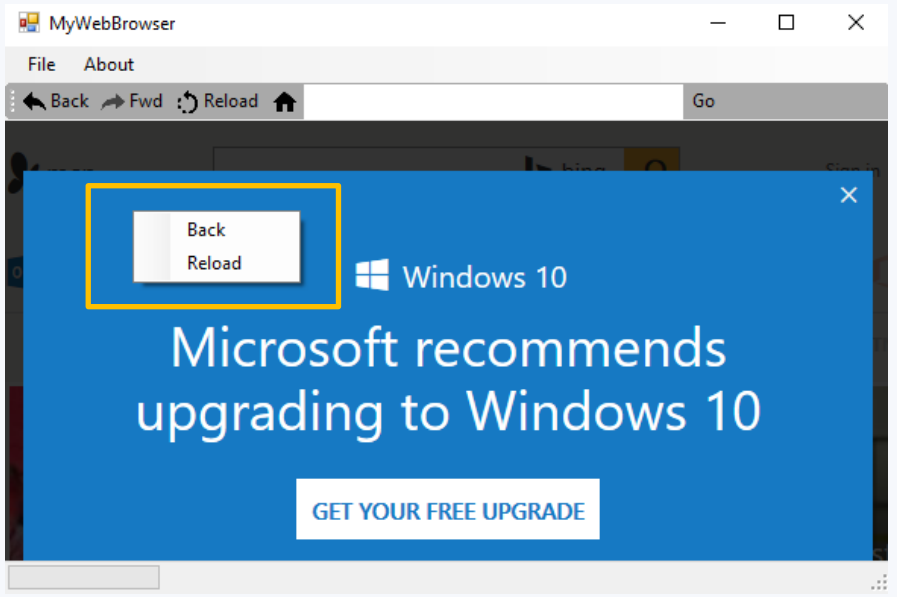
- System.Windows.Forms.**MenuStrip**

- The MenuStrip control represents the container for the menu structure of a form. You can add ToolStripMenuItem objects to the MenuStrip that represent the individual menu commands in the menu structure.

- Each ToolStripMenuItem can be a command for your application or a parent menu for other submenu items.

- Further reading: link

ContextMenuStrip



ContextMenuStrip

- System.Windows.Forms.ContextMenuStrip

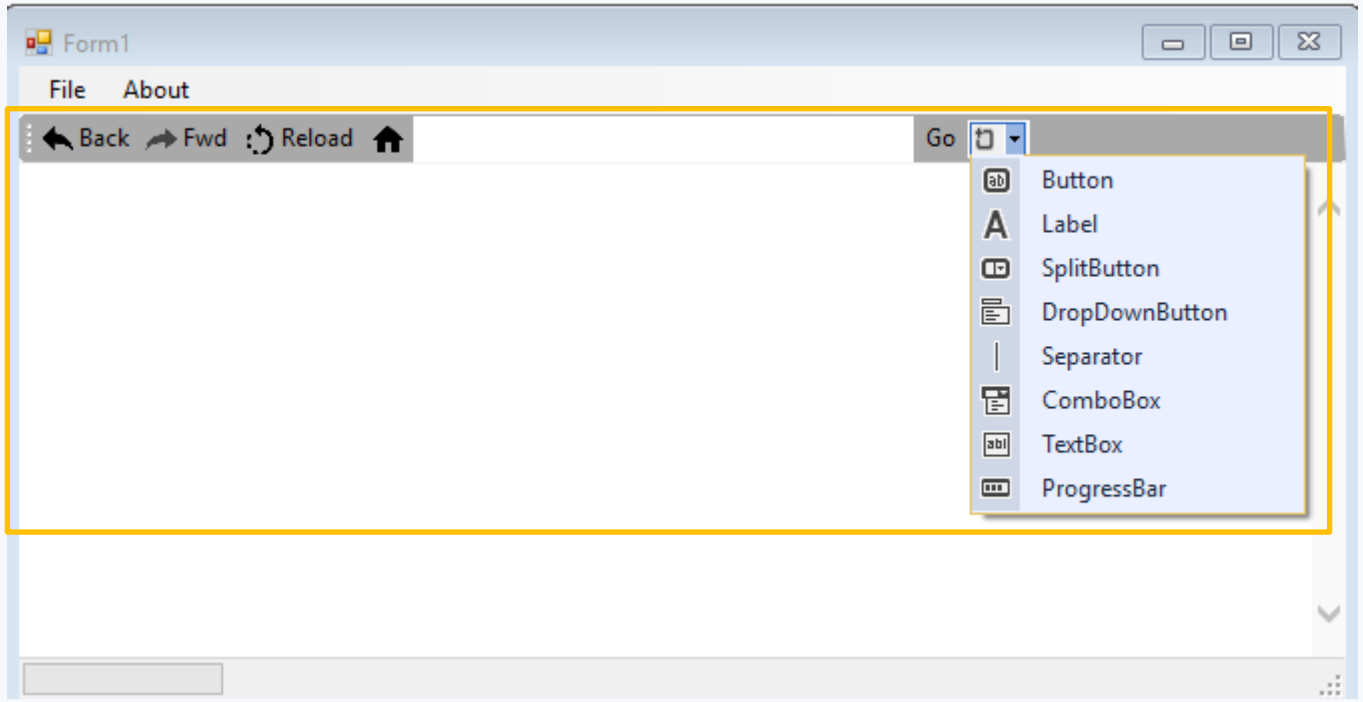
- Shortcut menus, also called context menus, appear at the mouse position when the user clicks the right mouse button. Shortcut *menus* provide options for the client area or the control at the mouse pointer location.

- Further reading: link

Demo

Add a context menu to the SimpleWebBrowser sample

ToolStrip


Form1
File
About
Back
Fwd
Reload
Go
Button
Label
SplitButton
DropDownButton
Separator
ComboBox
TextBox
ProgressBar

ToolStrip

- System.Windows.Forms.**ToolStrip**

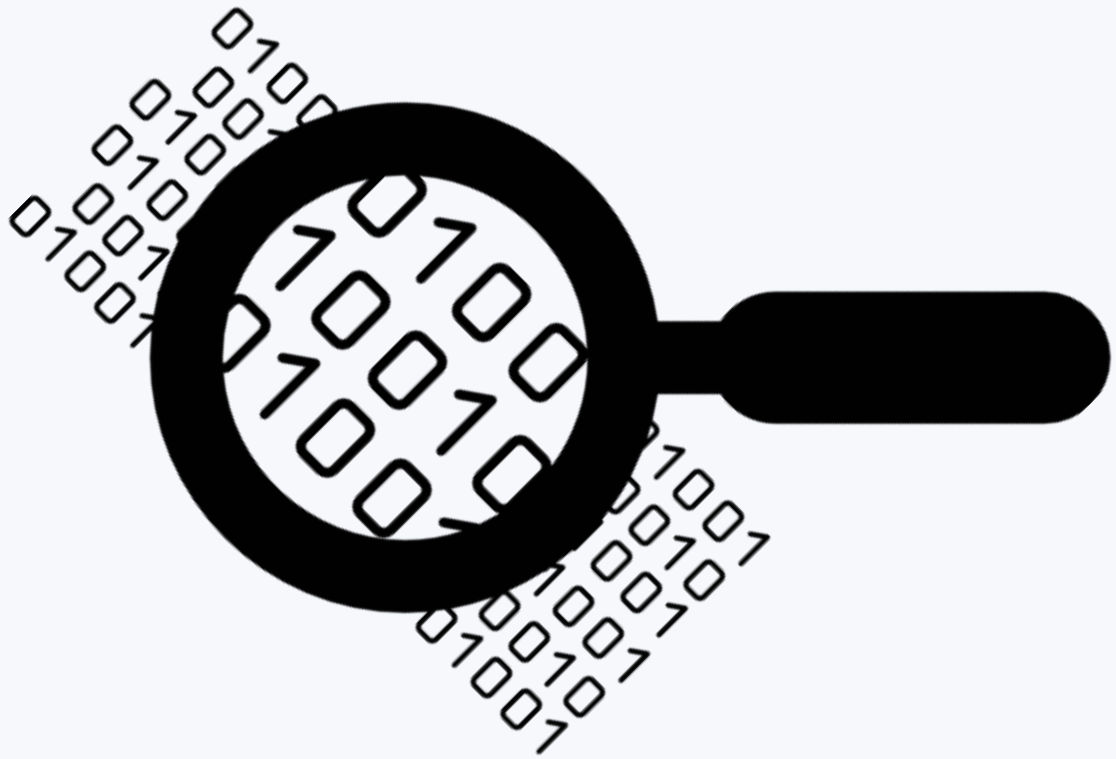
- The Windows Forms ToolStrip control and its associated classes provide a common framework for combining user interface elements into toolbars, status bars, and menus.

- ToolStrip controls offer a rich design-time experience that includes in-place activation and editing, custom layout, and rafting, which is the ability of toolbars to share horizontal or vertical space.

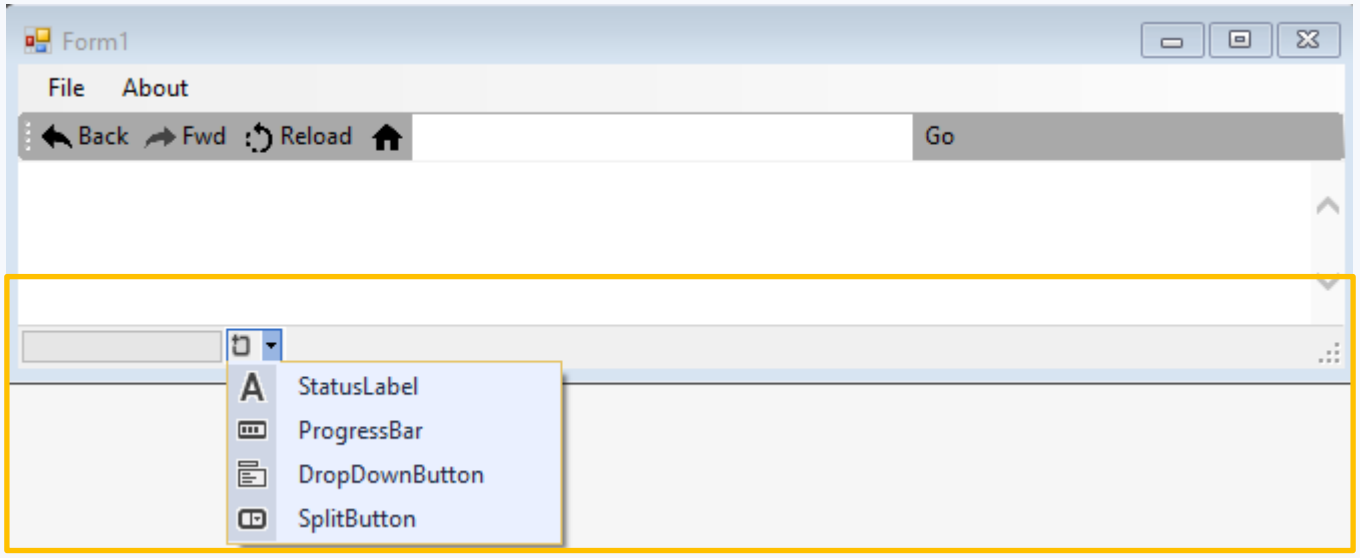
- Further reading: link

Demo

- Reimplement the SimpleWebBrowser sample using the ToolStripContainer



StatusStrip



StatusStrip

- Windows.Forms.**StatusStrip**

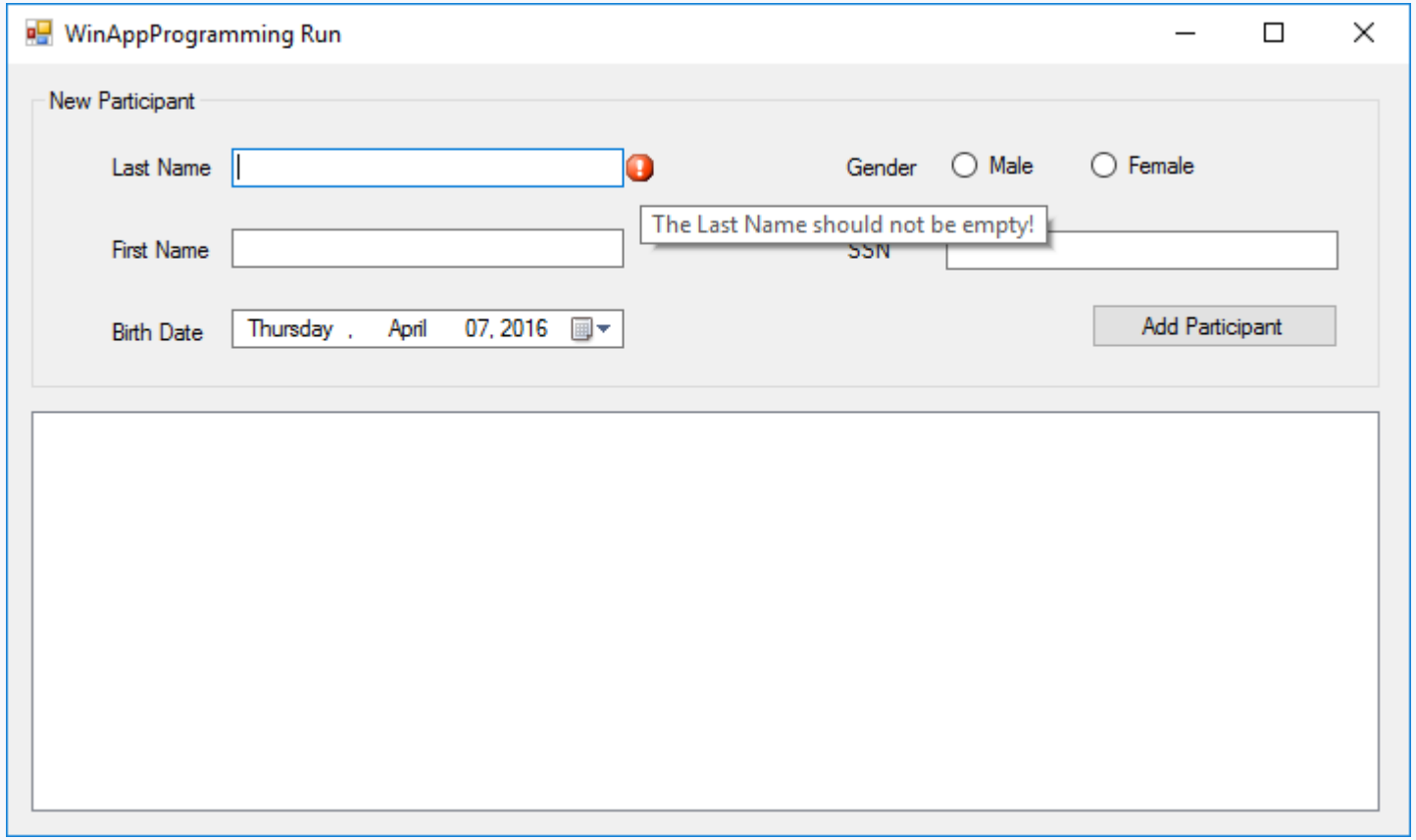
- A StatusStrip control displays information about an object being viewed on a Form, the object's components, or contextual information that relates to that object's operation within your application.

- Typically, a StatusStrip control consists of ToolStripStatusLabel objects, each of which displays text, an icon, or both. The StatusStrip can also contain ToolStripDropDownButton, ToolStripSplitButton, and ToolStripProgressBar controls.

- Further reading: link

Data Validation

Using **Validating** and **Validated** Events



Control.**Validating** Event

- Occurs when the control is validating.

- If the CausesValidation property is set to **false**, the Validating and Validated events are suppressed.

```
private void tbLastName_Validating(object sender, CancelEventArgs e)
{
        string lastName = ((TextBox) sender).Text.Trim();

        if (string.IsNullOrWhiteSpace(lastName))
        {
                e.Cancel = true;

                errorProvider.SetError((Control)sender, "Last Name is empty!");
        }
}
```

Control.**Validating** Event

- Events:
  1. Enter
  2. GotFocus
  3. Leave
  4. Validating
  5. Validated
  6. LostFocus

- If the Cancel property of the CancelEventArgs is set to **true** in the Validating event delegate, all events that would usually occur after the Validating event are suppressed.

Control.**Validated** Event

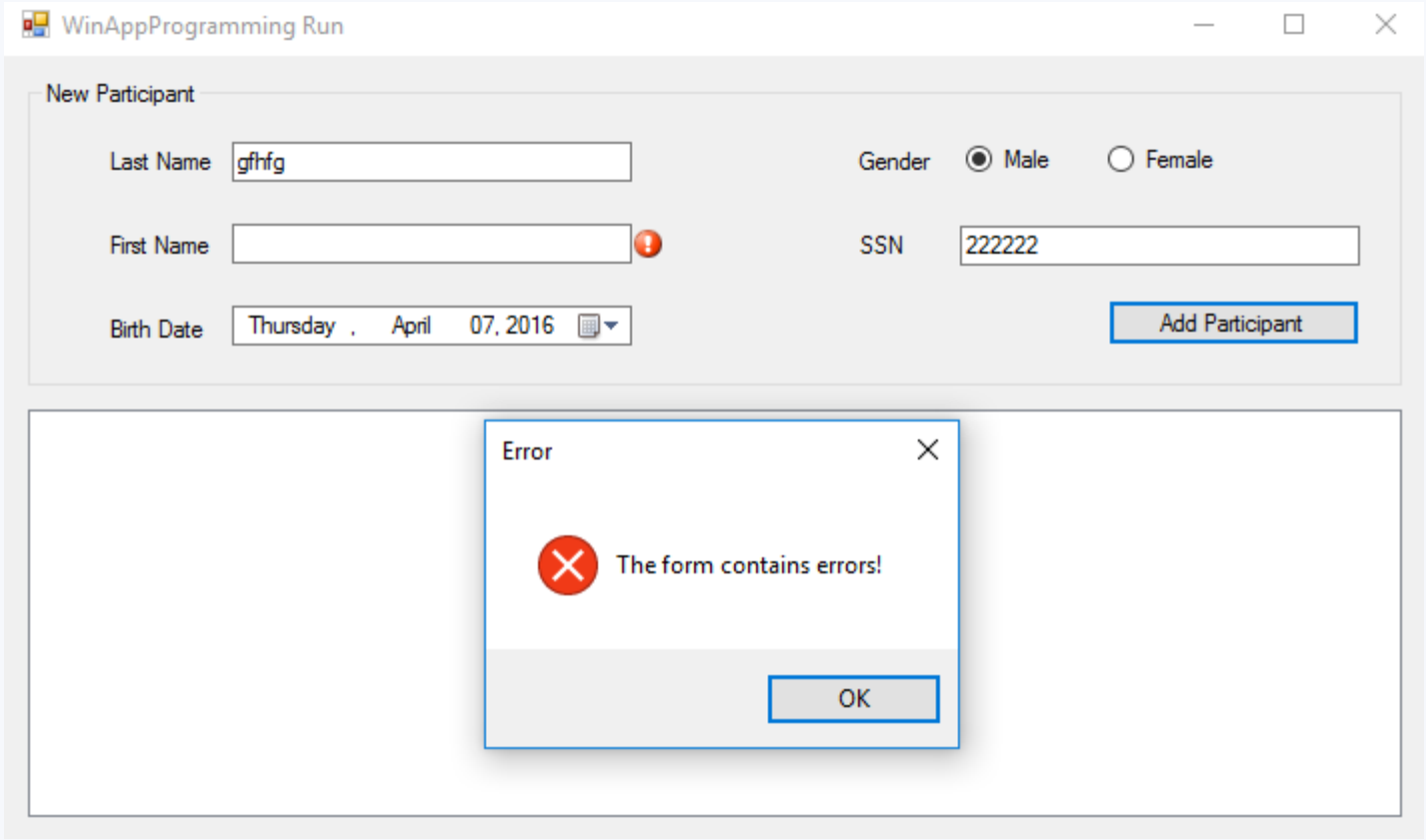
- Occurs when the control is finished validating.

```
private void control_Validated(object sender, EventArgs e)
{
        errorProvider.SetError((Control)sender, null);
}
```

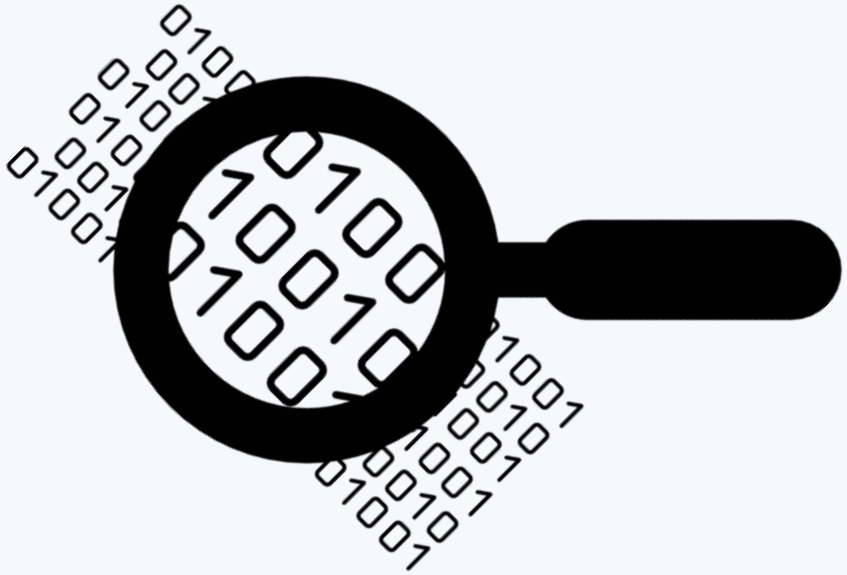
Validating the entire form

```
private void btnAdd_Click(object sender, EventArgs e){
        string firstName = tbFirstName.Text.Trim();
        string lastName = tbLastName.Text.Trim();

        bool formContainsErrors = false;

        if (string.IsNullOrWhiteSpace(lastName)){
                errorProvider.SetError(tbFirstName, "Last Name is empty!");
               formContainsErrors = true;
        }

        if (string.IsNullOrWhiteSpace(firstName)){…     }

        if (formContainsErrors)     {
               MessageBox.Show("The form contains errors!", "Error",
                      MessageBoxButtons.OK, MessageBoxIcon.Error);
        }
}
```

Validating the entire form



Demo



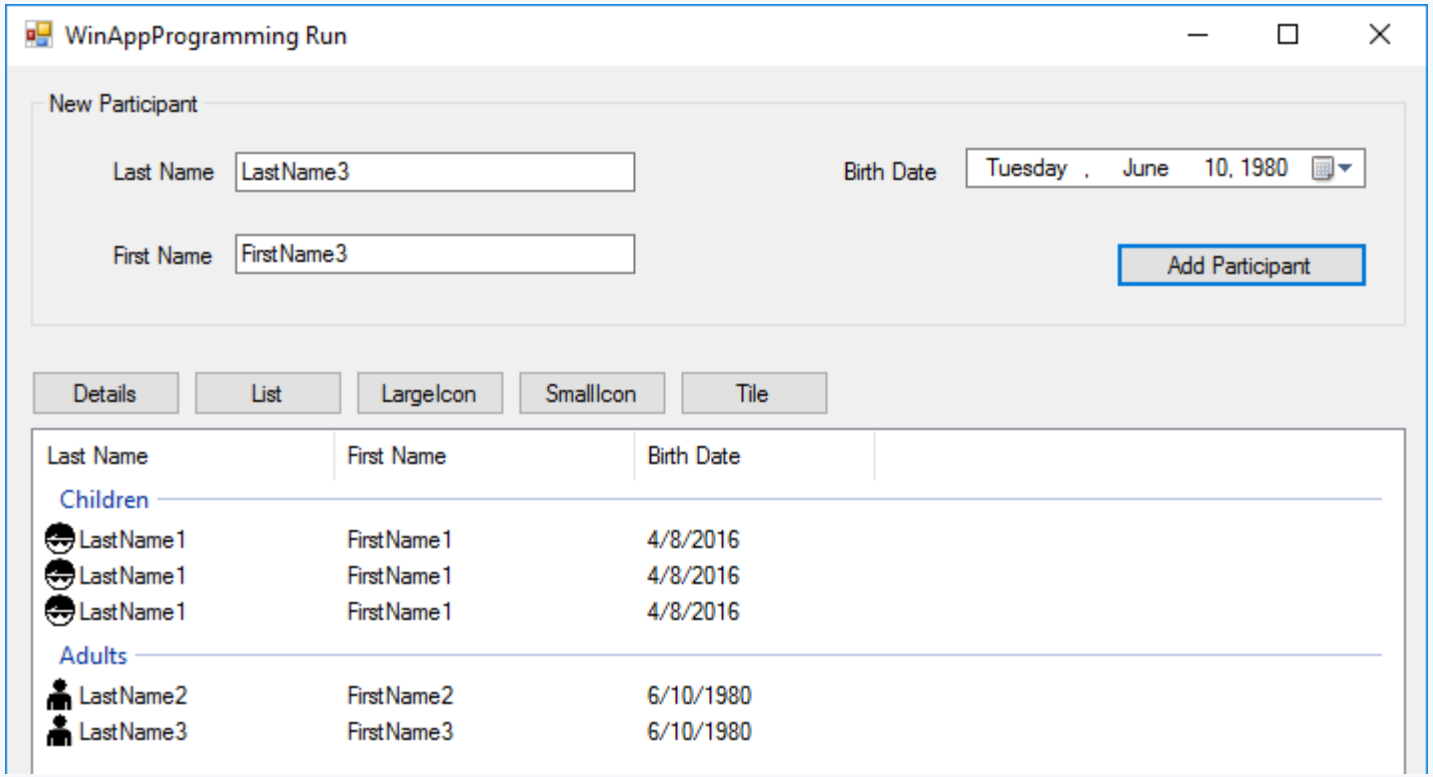
- Implement validations for an example containing only the first name and the last name fields.

Complex Visualization Controls

ListView

- System.Windows.Forms. ListView

- Represents a Windows list view control, which displays a collection of items that can be displayed using one of four different views.



ListView

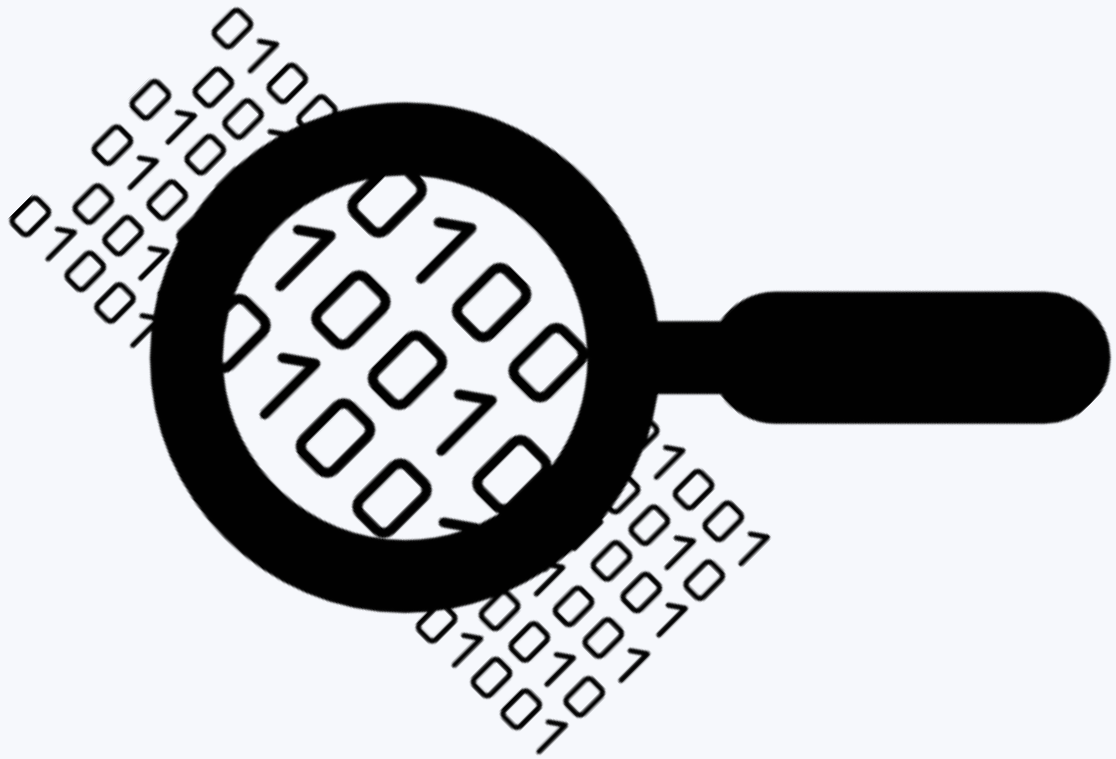
- Important properties:
  - Items
  - Columns
  - Groups
  - CheckBoxes
  - **View:** Details, List, Tile, SmallIcon, LargeIcon
  - **FullRowSelect:** True/False
  - **GridLines:** True/False
  - **SmallImageList** / **LargeImageList**

ListView

- Important events:
  - SelectedIndexChanged

Demo

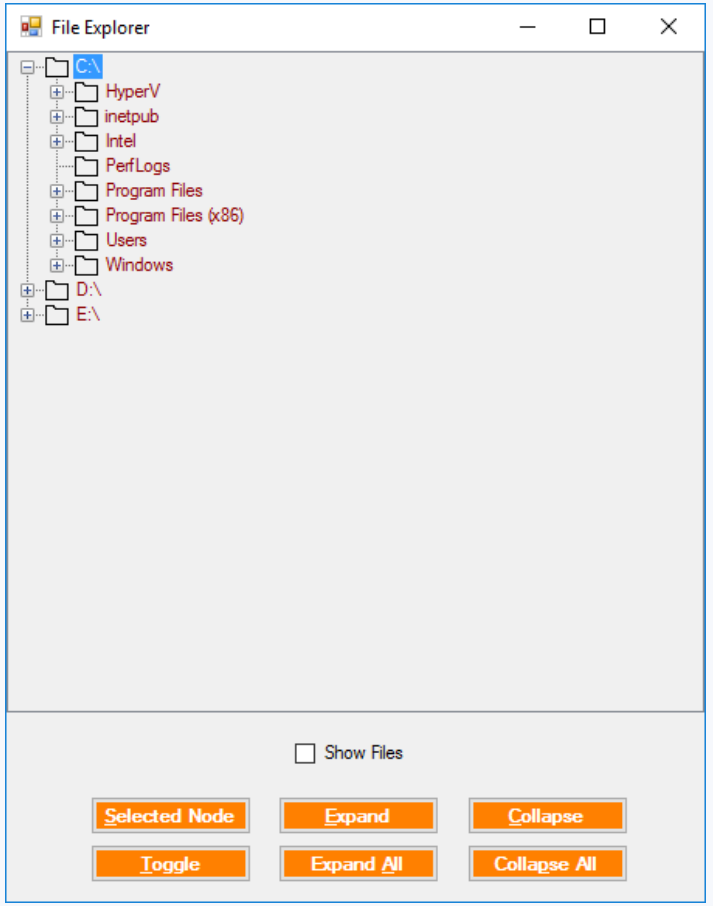
- Add a listview to the previous demo, similar to the ListViewBasicSample
- Check the ListViewSample



TreeView

- System.Windows.Forms.**TreeView**

- Displays a hierarchical collection of labeled items, each represented by a TreeNode.



TreeView

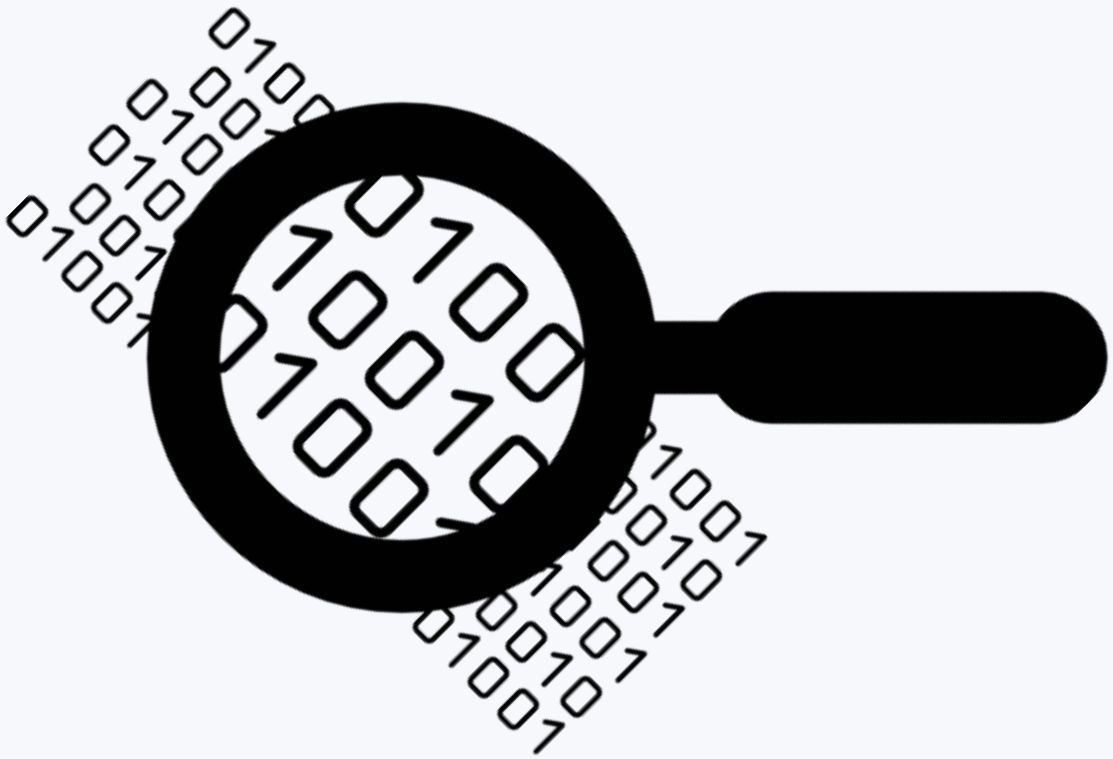
- Important properties:
  - Nodes
  - SelectedNode
  - CheckBoxes
  - ImageList

TreeView

- Important Methods:
  - Sort()
- Important Events:
  - AfterSelect
  - AfterExpand

Demo

- Check the TreeViewSample



Exception Handling

Try, Catch, Finally

- A **try** statement specifies a code block subject to error-handling or cleanup code. The **try** ***block*** must be followed by a **catch** ***block***, a **finally** ***block***, or both.
- The **catch** block executes when an error occurs in the **try** block.
- The **finally** block executes after execution leaves the **try** block (or if present, the **catch** block), to perform cleanup code, **whether or not an error occurred**.

Try, Catch, Finally

```
try
{
... // exception may get thrown within execution of this block
}
catch (ExceptionA ex)
{
... // handle exception of type ExceptionA
}
catch (ExceptionB ex)
{
... // handle exception of type ExceptionB
}
finally
{
... // cleanup code
}
```

Try, Catch, Finally

```
try{
    using (StreamReader sr = File.OpenText("data.txt"))
    {
        Console.WriteLine($"The first line of this file is {sr.ReadLine()}");
    }
}
catch (FileNotFoundException e){
    Console.WriteLine($"The file was not found: '{e}'");
}
catch (DirectoryNotFoundException e){
    Console.WriteLine($"The directory was not found: '{e}'");
}
catch (IOException e){
    Console.WriteLine($"The file could not be opened: '{e}'");
}
```

- Example from: https://docs.microsoft.com/en-us/dotnet/standard/exceptions/how-to-use-the-try-catch-block-to-catch-exceptions

Common Exception Types

- System.ArgumentException - Thrown when a function is called with a bogus argument.

- System.ArgumentNullException - Subclass of ArgumentException that's thrown when a function argument is (unexpectedly) null.

- System.ArgumentOutOfRangeException - Subclass of ArgumentException that's thrown when a (usually numeric) argument is too big or too small. For example, this is thrown when passing a negative number into a function that accepts only positive values.

Common Exception Types

- System.InvalidOperationException - Thrown when the state of an object is unsuitable for a method to successfully execute, regardless of any particular argument values. Examples include reading an unopened file or getting the next element from an enumerator where the underlying list has been modified partway through the iteration.

- System.NotSupportedException - Thrown to indicate that a particular functionality is not supported. A good example is calling the Add method on a collection for which IsReadOnly returns true.

- System.NotImplementedException - Thrown to indicate that a function has not yet been implemented.

Common Exception Types

- A catch block has access to an Exception object that contains information about the error. You use a catch block to either compensate for the error or *rethrow* the exception. You rethrow an exception if you merely want to log the problem, or if you want to rethrow a new, higher-level exception type.

- A finally block adds determinism to your program: the CLR endeavors to always execute it. It's useful for cleanup tasks such as closing network connections.

Demo

- Handle the exceptions for a simple calculator, similar to the one in the StandardExceptions sample

Custom exceptions

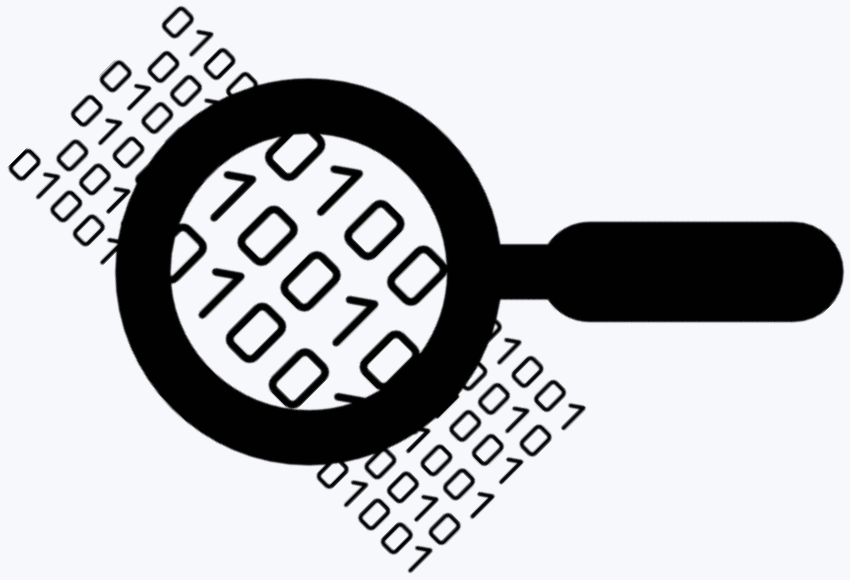
- The intrinsic exception types the CLR provides, coupled with custom messages will often be all you need to provide extensive information to a catch block when an exception is thrown.

- There will be times, however, when you want to provide more extensive information or need special capabilities in your exception. It is a trivial matter to create your own *custom exception* class; the only restriction is that it must derive (directly or indirectly) from System. Exception.

Custom exceptions

```
public class InvalidBirthDateException : Exception
{
    public DateTime BirthDate { get; set; }

    public InvalidBirthDateException(DateTime birthDay)
    {
        BirthDate = birthDay;
    }

   public override string Message
    {
        get
        {
            return "The birthDate " + BirthDate + " is invalid";
        }
    }
}
```

Demo



- Implement an example that uses the InvalidBirthDateException;
- Check the ValidationCustomExceptions sample.

Further reading

- MSDN Best Practices for Exceptions: https://docs.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions

Streams

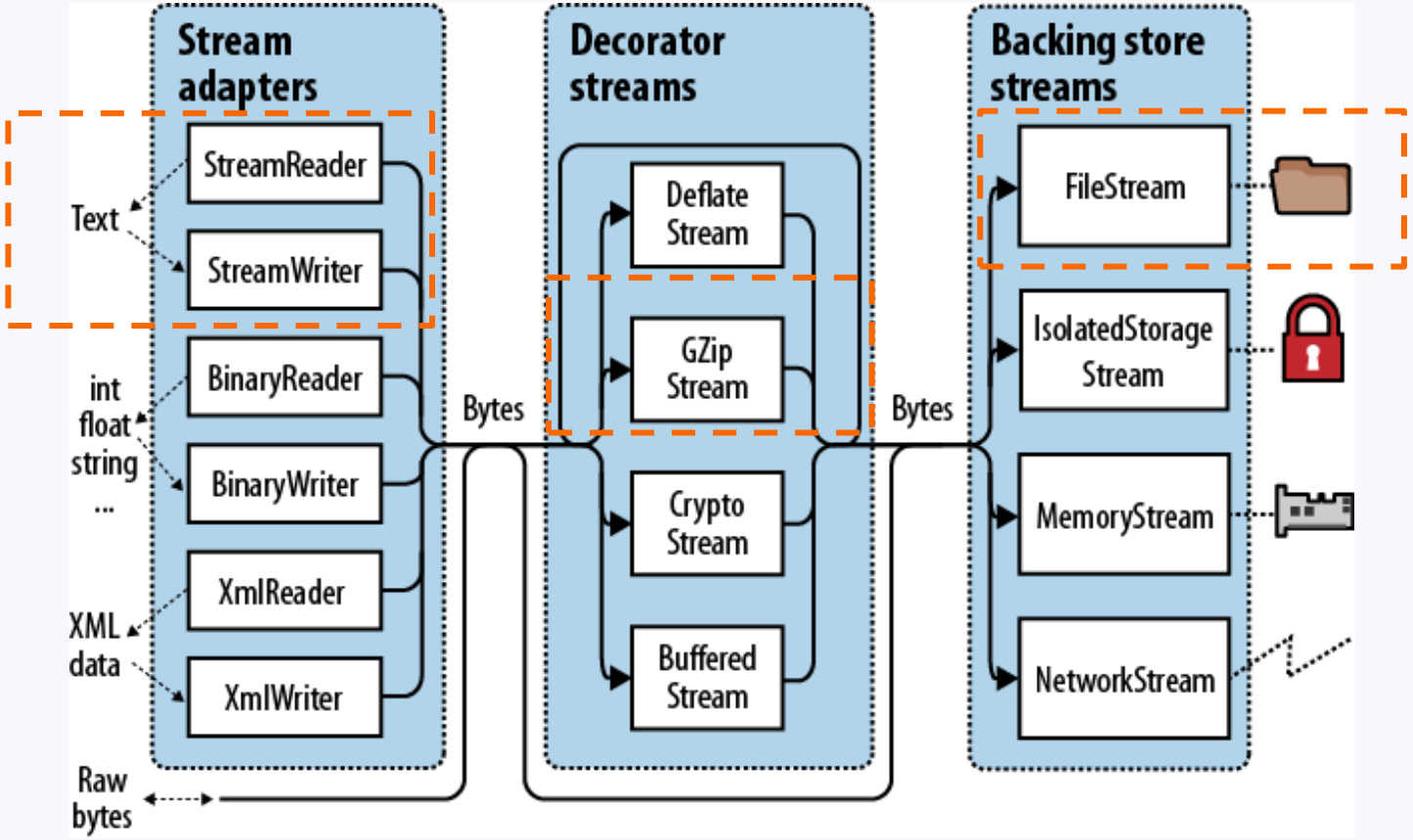
Stream Architecture

- A **stream** is an abstraction of a **sequence of bytes**, such as a **file**, an **input/output device**, an inter-process communication pipe, or a **TCP/IP socket**.
- System.IO.**Stream** class and its derived classes provide a generic view of these different types of input and output, and isolate the programmer from the specific details of the operating system and the underlying devices.
- System.IO.**Stream** is the abstract base class of all streams.
- Further reading: link

Stream Architecture

- Unlike an array, where all the backing data exists in memory at once, a stream deals with data serially—either one byte at a time or in blocks of a manageable size. Hence, a stream can use little memory regardless of the size of its backing store.

Stream Architecture



System.IO.FileStream

- Can be used to read from, write to, open, and close files on a file system.

```
var fs = new FileStream (
                    "x.bin",
                    FileMode.Open,
                    FileAccess.Read
                    ))
```

- Further reading: link

System.IO.FileMode

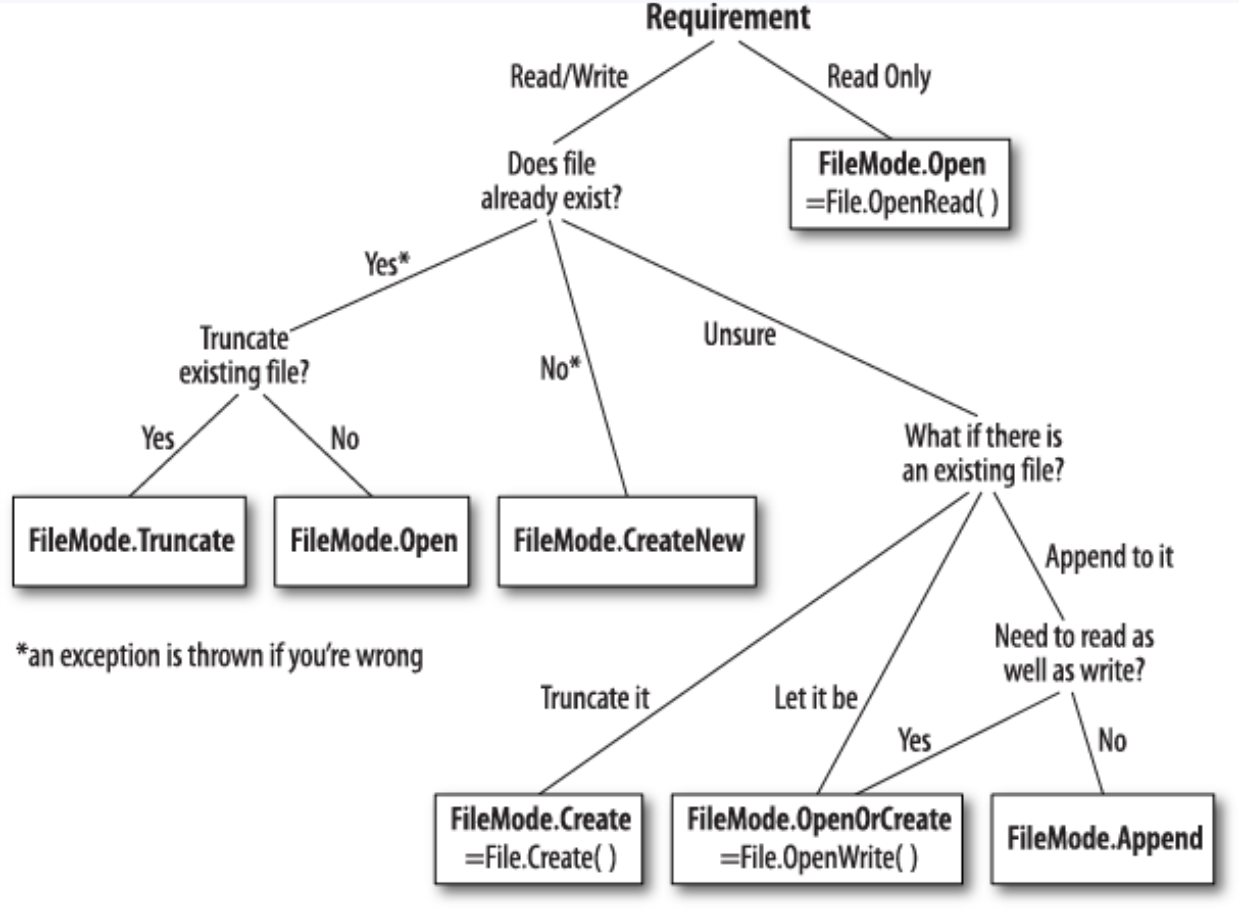
Append	Opens the file if it exists and seeks to the end of the file, or creates a new file. This requires FileIOPermissionAccess.Append permission.
Create	Specifies that the operating system should create a new file. If the file already exists, it will be overwritten. This requires FileIOPermissionAccess.Write permission.
CreateNew	Specifies that the operating system should create a new file. This requires FileIOPermissionAccess.Write permission.
Open	Specifies that the operating system should open an existing file. The ability to open the file is dependent on the value specified by the FileAccess enumeration.

System.IO.FileMode

OpenOrCreate	Specifies that the operating system should open a file if it exists; otherwise, a new file should be created. If the file is opened with FileAccess.Read, FileIOPermissionAccess.Read permission is required. If the file access is FileAccess.Write, FileIOPermissionAccess.Write permission is required. If the file is opened with FileAccess.ReadWrite, both FileIOPermissionAccess.Read and FileIOPermissionAccess.Write permissions are required.
Truncate	Specifies that the operating system should open an existing file. When the file is opened, it should be truncated so that its size is zero bytes. This requires FileIOPermissionAccess.Write permission.

- Further reading: link

System.IO.FileMode



System.IO.Compression.**GZipStream**

- Provides methods and properties used to compress and decompress streams by using the GZip data format specification.

- Sample: https://docs.microsoft.com/en-us/dotnet/api/system.io.compression.gzipstream

System.IO.Compression.GZipStream

- Compressing all the files in a directory:

```
foreach (FileInfo fileToCompress in directorySelected.GetFiles())
{
      using (FileStream originalFileStream = fileToCompress.OpenRead())
      {
            using (FileStream fs = File.Create(fileToCompress.FullName + ".gz"))
            {
                  using (GZipStream compressionStream = new GZipStream(fs,
                     CompressionMode.Compress))
                  {
                        originalFileStream.CopyTo(compressionStream);
                  }
            }
      }
}
```

System.IO.FileAccess

Read	Read access to the file. Data can be read from the file. Combine with Write for read/write access.
ReadWrite	Read and write access to the file. Data can be written to and read from the file.
Write	Write access to the file. Data can be written to the file. Combine with Read for read/write access.

- Further reading: link

Writing Text Files

- System.IO.**StreamWriter**
  - derived from the abstract TextWriter class;
  - designed for writing characters to a stream in a particular encoding.

- Useful Methods:
  - Write
  - WriteLine

- Further reading: link

Reading Text Files

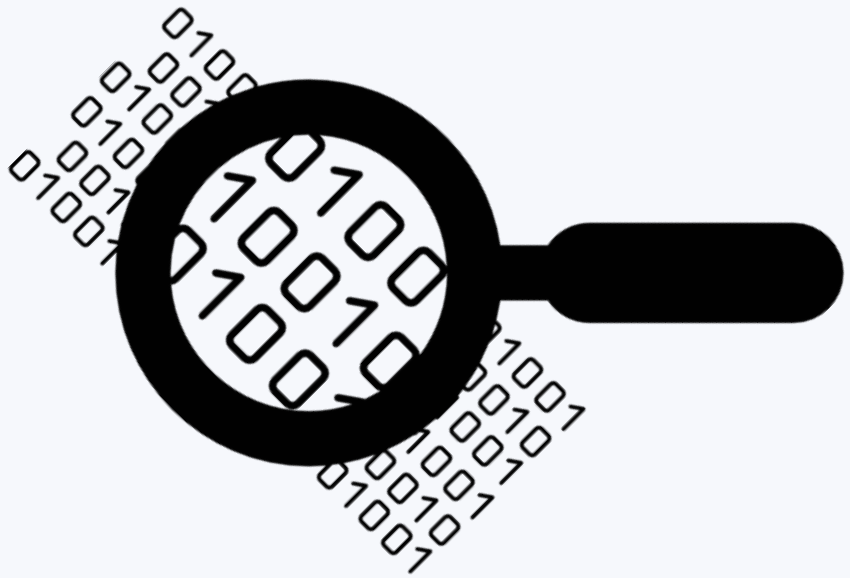
- System.IO.**StreamReader**
  - derived from the abstract TextReader class;
  - designed for reading characters from a stream in a particular encoding.

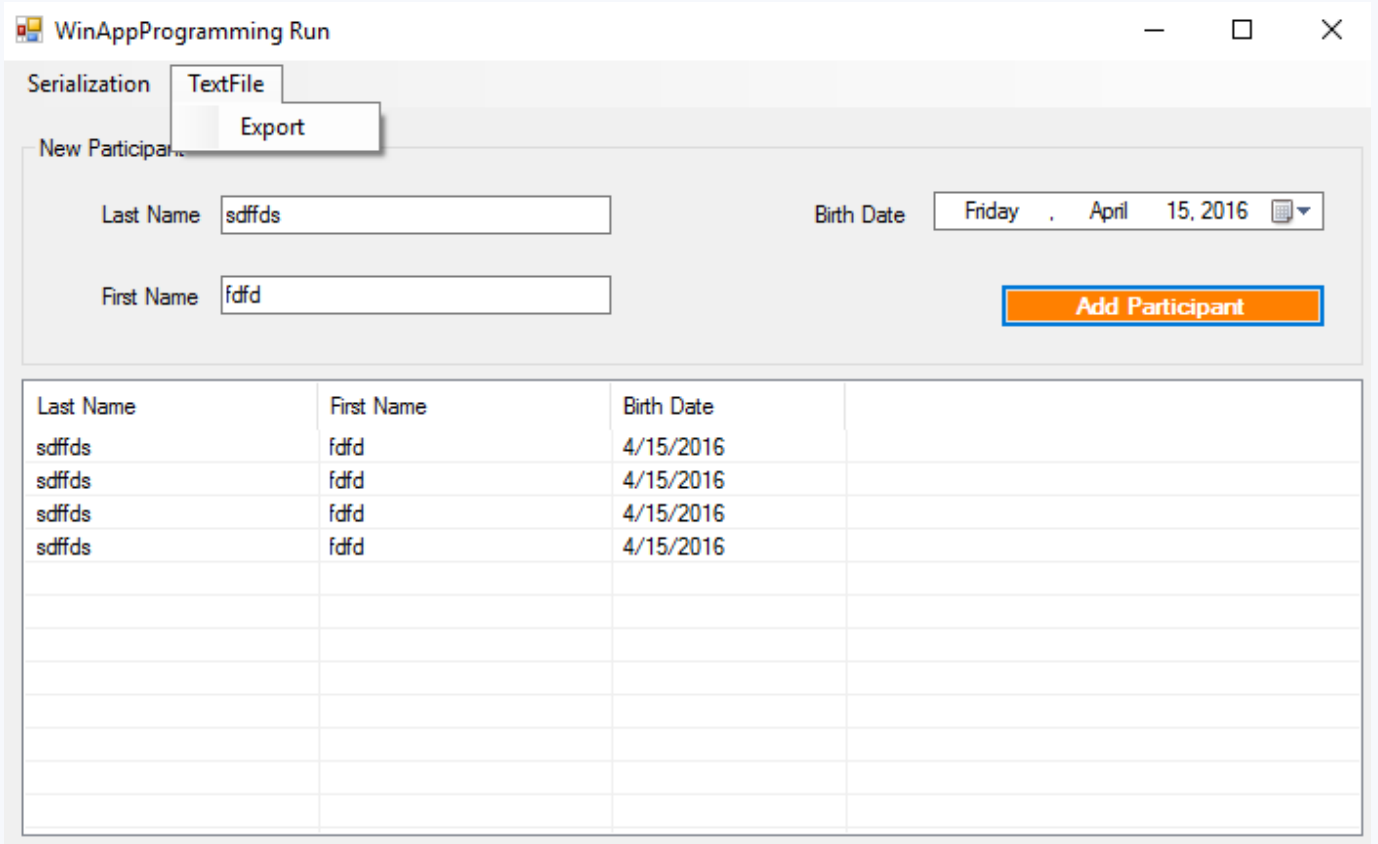
- Useful Methods:
  - Read / ReadAsync
  - ReadLine / ReadLineAsync

- Further reading: link

```
static void Main(string[] args){
        // Get the directories currently on the C drive.
        DirectoryInfo[] cDirs = new DirectoryInfo(@"c:\").GetDirectories();

        // Write each directory name to a file.
        using (StreamWriter sw = new StreamWriter("CDriveDirs.txt")) {
                foreach (DirectoryInfo dir in cDirs){
                        sw.WriteLine(dir.Name);
                }
        }

        // Read and show each line from the file.
        string line = "";
        using (StreamReader sr = new StreamReader("CDriveDirs.txt")) {
                while ((line = sr.ReadLine()) != null)                  {
                        Console.WriteLine(line);
                }
        }
}
```

Demo




WinAppProgramming Run
Serialization
TextFile
Export
New Participant
Last Name
sdffds
First Name
fdfd
Birth Date
Friday , April 15, 2016
Add Participant
Last Name
First Name
Birth Date
sdffds fdfd 4/15/2016
sdffds fdfd 4/15/2016
sdffds fdfd 4/15/2016
sdffds fdfd 4/15/2016

File and Directory Operations

- The **System.IO** namespace provides a set of types for performing "utility" file and directory operations, such as **copying** and **moving**, **creating directories**, and **setting file attributes** and **permissions**.

- Static classes:
  - System.IO.**File** and System.IO.**Directory**

- Instance method classes (constructed with a file or directory name):
  - System.IO.**FileInfo** and System.IO.**DirectoryInfo**

- Additional class:
  - System.IO.**Path**

Special Folders

```
string myDocPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);
```

- **Examples:** Desktop, ProgramFiles, ProgramFilesX86, Favorites, MyMusic, MyVideos
- All special folders: link

System.IO.File

- Important methods:

```
bool Exists (string path); // Returns true if the file is present

void Delete (string path);
void Copy (string sourceFileName, string destFileName);
void Move (string sourceFileName, string destFileName);
void Replace (string sourceFileName, string destinationFileName,
string destinationBackupFileName);

FileAttributes GetAttributes (string path);
void SetAttributes (string path, FileAttributes fileAttributes);
```

System.IO.Directory

- Important methods:

```
void SetCurrentDirectory (string path);
DirectoryInfo CreateDirectory (string path);
DirectoryInfo GetParent (string path);
string GetDirectoryRoot (string path);
string[] GetLogicalDrives();
// The following methods all return full paths:
string[] GetFiles (string path);
string[] GetDirectories (string path);
string[] GetFileSystemEntries (string path);
IEnumerable<string> EnumerateFiles (string path);
IEnumerable<string> EnumerateDirectories (string path);
```

System.IO.FileInfo

- Provides additional properties such as:
  - Extension,
  - Length,
  - IsReadOnly,
  - Directory.

- Further reading: link

Other classes

- System.IO.**DirectoryInfo**
  - Further reading: link

- System.IO.**Path**
  - Further reading: link

Serialization

Serialization and Deserialization

- *Serialization* is the act of taking an in-memory object or *object graph* (set of objects that reference each other) and flattening it into a stream of bytes or XML nodes that can be stored or transmitted.

- *Deserialization* works in reverse, taking a data stream and reconstituting it into an in-memory object or object graph.

- Serialization and deserialization are typically used to:
  - Transmit objects across a network or application boundary.
  - Store representations of objects within a file or database.

Serialization Engines

- There are four main serialization mechanisms in the .NET Framework:
  - The data contract serializer
  - The binary serializer
  - The (attribute-based) XML serializer (XmlSerializer)
  - The IXmlSerializable interface

Binary Serializer

```
[Serializable]
public sealed class Person
{
        public string Name;
        public int Age;
}
```

- The [Serializable] attribute instructs the serializer to include all fields in the type. This includes both private and public fields (but not properties). Every field must itself be serializable; otherwise, an exception is thrown.

- Primitive .NET types such as string and int support serialization (as do many other .NET types).

Binary Serializer

```
private void btnSerialize_Click(object sender, EventArgs e){
        IFormatter formatter = new BinaryFormatter();
        using (FileStream s = File.Create("serialized.bin"))
                formatter.Serialize(s, _participants);
}

private void btnDeserialize_Click(object sender, EventArgs e){
        IFormatter formatter = new BinaryFormatter();
        using (FileStream s = File.OpenRead("serialized.bin"))        {
                _participants = (List<Participant>)formatter.Deserialize(s);
                DisplayParticipants();
        }
}
```

Binary Serialization Attributes

- [NonSerialized]
  - Fields that should not be serialized, such as those used for temporary calculations, or for storing file or window handles, must be marked explicitly with the [NonSerialized] attribute

```
public sealed class Person
{
        public string Name;
        public DateTime DateOfBirth;
        [NonSerialized]
        public int Age;
        [NonSerialized]
        public bool Valid = true;
        public Person() { Valid = true; }
}
```

Dialogs

Modal & Modeless

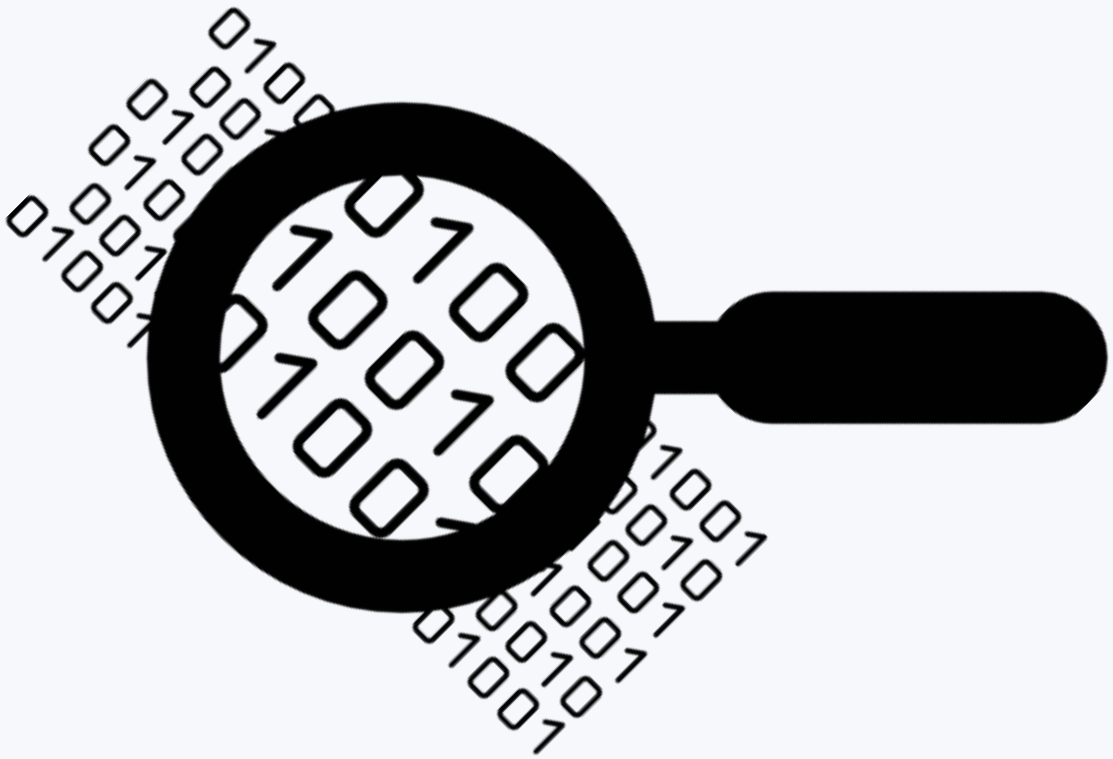
- Forms can exhibit either **modal** or **modeless** behavior
- A **modal form** is one that demands the user's immediate attention, and blocks input to any other windows the application may have open.
- Good UI design suggests that you should use **modal** dialogs **only** when absolutely necessary. Typical examples: error messages

Modal & Modeless

```
//modal
Form frm = new Form(  );
frm.ShowDialog(  );

//modeless
Form frm = new Form(  );
frm.Show (  );
```

Modal & Modeless

- Implement an example similar to the DialogSample project



Useful Properties

Property	Type	Description
AcceptButton	IButtonControl	Gets or sets the button on a form that is clicked when the user presses Enter, irrespective of which control actually has focus.
CancelButton	IButtonControl	Gets or sets the button on a form that is clicked when the user presses Escape, irrespective of which control actually has focus.
ControlBox	Boolean	Gets or sets a value indicating if the system menu can be displayed and if the Close button (X) is displayed on the right end of the titlebar. The system menu is exposed by clicking on the icon in the titlebar if the Icon property is set, or by right-clicking on the titlebar if the Icon property is not set. Default is true.
DialogResult	DialogResult	Gets or sets the return value from a form or dialog box displayed modally. Legal values are members of the DialogResult enumeration. This property is covered in detail on the following slide.
FormBorderStyle	FormBorderStyle	Gets or sets the border style. In addition to appearance, the FormBorderStyle dictates whether or not the form will be resizable.
Icon	Icon	Gets or sets the icon for the form.
ShowInTaskBar	Boolean	Gets or sets a value indicating if the form Text property is displayed in the Windows taskbar. Default is true.

Useful Properties

Property	Type	Description
Size	Size	Gets or sets both the height and width of the form.
StartPosition	FormStartPosition	Gets or sets the starting position of the form.
TopMost	Boolean	Gets or sets value indicating the form is displayed on top all the other forms, even if it is not the active form. Typically used for modeless dialog boxes, which should always be visible.

Useful Properties > FormBorderStyle

Value	Description
Fixed3D	Nonresizable, 3-D border.
FixedDialog	Nonresizable, dialog-style border
FixedSingle	Nonresizable, single line border
FixedToolWindow	Nonresizable, tool window border
None	No border
Sizable	Default value. Standard resizable border
SizableToolWindow	Resizable, tool window border

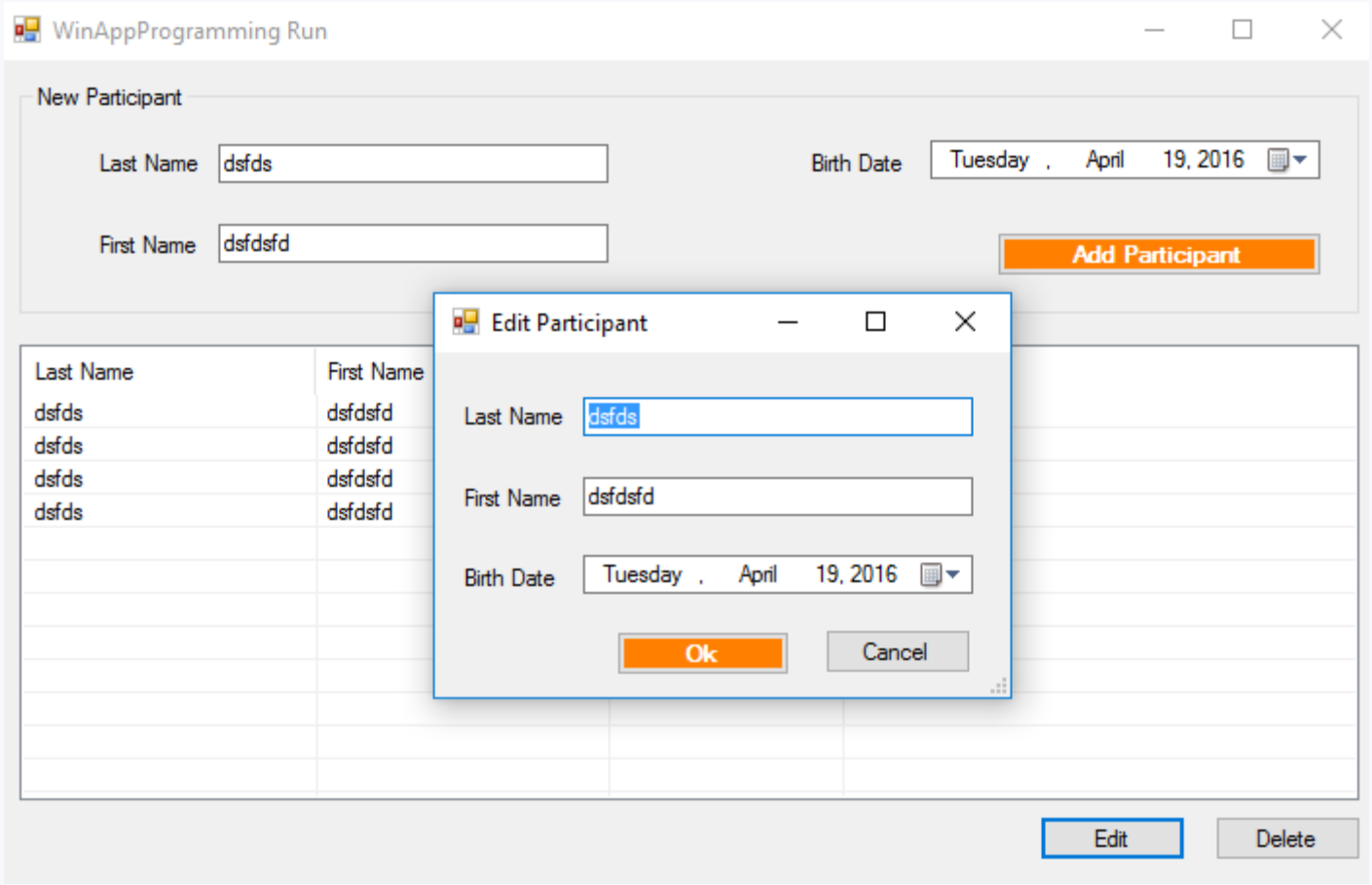
Useful Properties > DialogResult

Value	Description
Abort	Abort
Cancel	Cancel
Ignore	Ignore
No	No
None	Returns nothing. Modal dialog is not terminated.
OK	OK
Retry	Retry
Yes	Yes

Useful Properties > StartPosition

Value	Description
CenterParent	Center form within parent form
CenterScreen	Center form within current display
Manual	Location and size of form dictates its starting position
WindowsDefaultBounds	Form at Windows default location with bounds determined by Windows default
WindowsDefaultLocation	Form at Windows default location with dimensions specified in form's constructor

Transferring data between forms



Transferring data between forms

```
EditForm editForm = new EditForm((Participant)lvParticipants.SelectedItems[0].Tag);
if (editForm.ShowDialog() == DialogResult.OK)
        DisplayParticipants();
```

```
public partial class EditForm : Form
{
        private readonly Participant _participant;

        public EditForm(Participant participant)    {
                _participant = participant;

                InitializeComponent();
        }
}
```

Transferring data between forms

CommonDialog Classes

- FileDialog
- ColorDialog
- FontDialog
- PageSetupDialog
- PrintDialog

Data Binding

Data Binding

- Data binding in Windows Forms gives you the means to display and make changes to information from a data source in controls on the form.
- You can bind to both traditional data sources as well as almost any structure that contains data.
- Further reading: link

Types of Data Binding

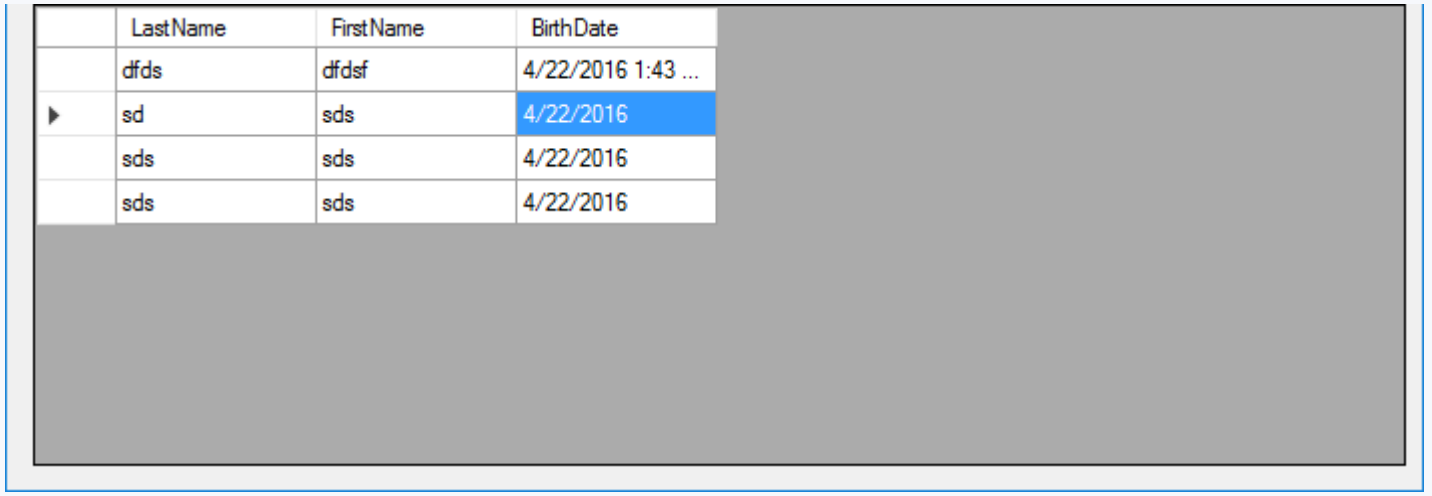
Type	Description
Simple data binding	The ability of a control to bind to a single data element, such as a value in a column in a dataset table. This is the type of binding typical for controls such as a TextBox control or Label control, which are controls that typically only displays a single value. In fact, any property on a control can be bound to a field in a database.
Complex data binding	The ability of a control to bind to more than one data element, typically more than one record in a database. Complex binding is also called list-based binding. Examples of controls that support complex binding are the DataGridView, ListBox, and ComboBox controls.

Simple data binding

```
control.DataBindings.Add(
        string propertyName,
        object dataSource,
        string dataMember,
        bool formattingEnabled,
        DataSourceUpdateMode dataSourceUpdateMode
);
```

```
tbLastName.DataBindings.Add("Text",_viewModel,"LastName“);
tbLastName.DataBindings.Add("Text",_viewModel,"LastName",false,DataSourceUpdateMode.OnPropertyChanged);
```

Complex data binding




```
dgvParticipants.DataSource = _viewModel.Participants;
```

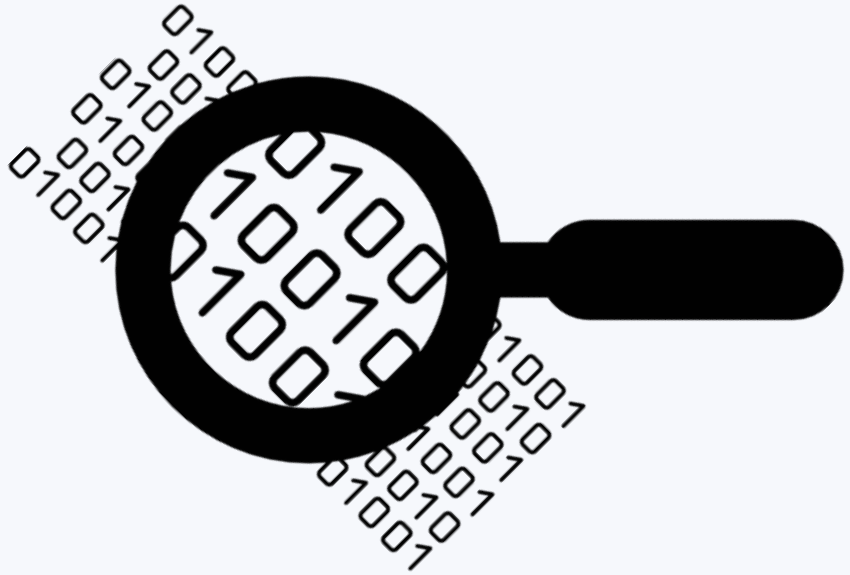
Change notification

- One of the most important concepts of Windows Forms data binding is ***change notification***. To ensure that your data source and bound controls always have the most recent data, you must add change notification for data binding.

- Specifically, you want to ensure that bound controls are notified of changes that were made to their data source, and the data source is notified of changes that were made to the bound properties of a control.

- Cases:
  - Simple Binding – INotifyPropertyChanged
  - Complex data binding - IBindingList

Demo



- Discuss the DataBinding sample

Demo

- Add a **UnitTest** project, similar to DataBindingSample.Tests

Databases

Data Access

- ADO.NET - Active Data Objects
- NHibernate
- Entity Framework

ADO.NET - Active Data Objects

- Approaches:
  - Connection Oriented Data Access Architecture - the application makes a connection to the Data Source and then interacts with it through SQL requests using the same connection. In these cases the application stays connected to the database system even when it is not using any Database Operations
  - Disconnected Data Access Architecture – the application can manipulate a set of DataTable objects (contained within a DataSet) that functions as a client-side copy of the external data. The connection is automatically opened and closed on when needed.
- Further reading: link

ADO.NET Data Providers

Type of Object	
Connection	Provides the ability to connect to and disconnect from the data store. Connection objects also provide access to a related transaction object.
Command	Represents a SQL query or a stored procedure. Command objects also provide access to the provider's data reader object.
DataReader	Provides forward-only, read-only access to data using a server-side cursor.
DataAdapter	Transfers DataSets between the caller and the data store. Data adapters contain a connection and a set of four internal command objects used to select, insert, update, and delete information from the data store.
Parameter	Represents a named parameter within a parameterized query.
Transaction	Encapsulates a database transaction.

ADO.NET Data Providers

- ADO.NET supports multiple *data providers*, each of which is optimized to interact with a specific DBMS

- Connection:
  - SqlConnection;
  - OracleConnection;
  - OdbcConnection;
  - MySqlConnection;
  - SqLiteConnection.

Connection

- The connection tells the rest of the ADO.NET code which database it is talking to.

- SQLiteConnection

```
SQLiteConnection conn = new SQLiteConnection("Data Source=database.db");
```

- SqlConnection

```
SqlConnection conn = new SqlConnection( "Data Source=DatabaseServer;
Initial Catalog=Northwind;User ID=YourUserID;Password=YourPassword");
```

Command

- The **Command** object specifies the type of interaction that will be performed on a database. For example, you can do **select**, **insert**, **update**, and **delete** commands on rows of data in a database table.

```
SQLiteCommand command = new SQLiteCommand("SELECT * FROM Participant",
_dbConnection);
```

Command

- ExecuteScalar()
  - Executes the query and returns the first column of the first row in the result set returned by the query. All other columns and rows are ignored.
  - If the value in the database is **null**, the query returns **DBNull.Value**.

```
// 1. Instantiate a new command
SqlCommand cmd = new SqlCommand("select count(*) from Participant", conn);

// 2. Call ExecuteScalar to send command
int count = (int)cmd.ExecuteScalar();
```

Command

- ExecuteNonQuery()
  - Can be used to change the data in a database by executing UPDATE, INSERT, or DELETE statements.

Command

```
const string stringSql = "DELETE FROM Participant WHERE Id=@id";
try{
        //Remove from the database
        _dbConnection.Open();
        SQLiteCommand sqlCommand = new SQLiteCommand(stringSql, _dbConnection);
        var idParameter = new SQLiteParameter("@id");
        idParameter.Value = participant.Id;
        sqlCommand.Parameters.Add(idParameter);

        sqlCommand.ExecuteNonQuery();
}
finally{
        if (_dbConnection.State != ConnectionState.Closed) _dbConnection.Close();
}
```

DataReader

- The ADO.NET DataReader can be used to retrieve a read-only, forward-only stream of data from a database. Results are returned as the query executes, and are stored in the network buffer on the client until you request them using the Read method of the DataReader.

DataReader

```
SQLiteDataReader sqlReader = sqlCommand.ExecuteReader();
try {
        while (sqlReader.Read())
        {
                _participants.Add(new Participant(
                        (long) sqlReader["Id"],
                        (string) sqlReader["LastName"],
                        (string) sqlReader["FirstName"],
                        DateTime.Parse((string) sqlReader["BirthDate"])));
        }
}
finally {
        // Always call Close when done reading.
        sqlReader.Close();
}
```

Transaction

```
try{
        _dbConnection.Open();
        dbCommand.Transaction = _dbConnection.BeginTransaction();

        participant.Id = (long)dbCommand.ExecuteScalar();

        dbCommand.Transaction.Commit();
}
catch (Exception){
        dbCommand.Transaction.Rollback();
        throw;
}
finally{
        if (_dbConnection.State != ConnectionState.Closed) _dbConnection.Close();
}
```

DataSet and DataAdapter

- A **DataSet** is an in-memory data store that can hold numerous tables (**DataTable**). DataSets only hold data and do not interact with a data source.

- It is the **DataAdapter** that manages connections with the data source and gives us disconnected behavior. The DataAdapter opens a connection only when required and closes it as soon as it has performed its task.

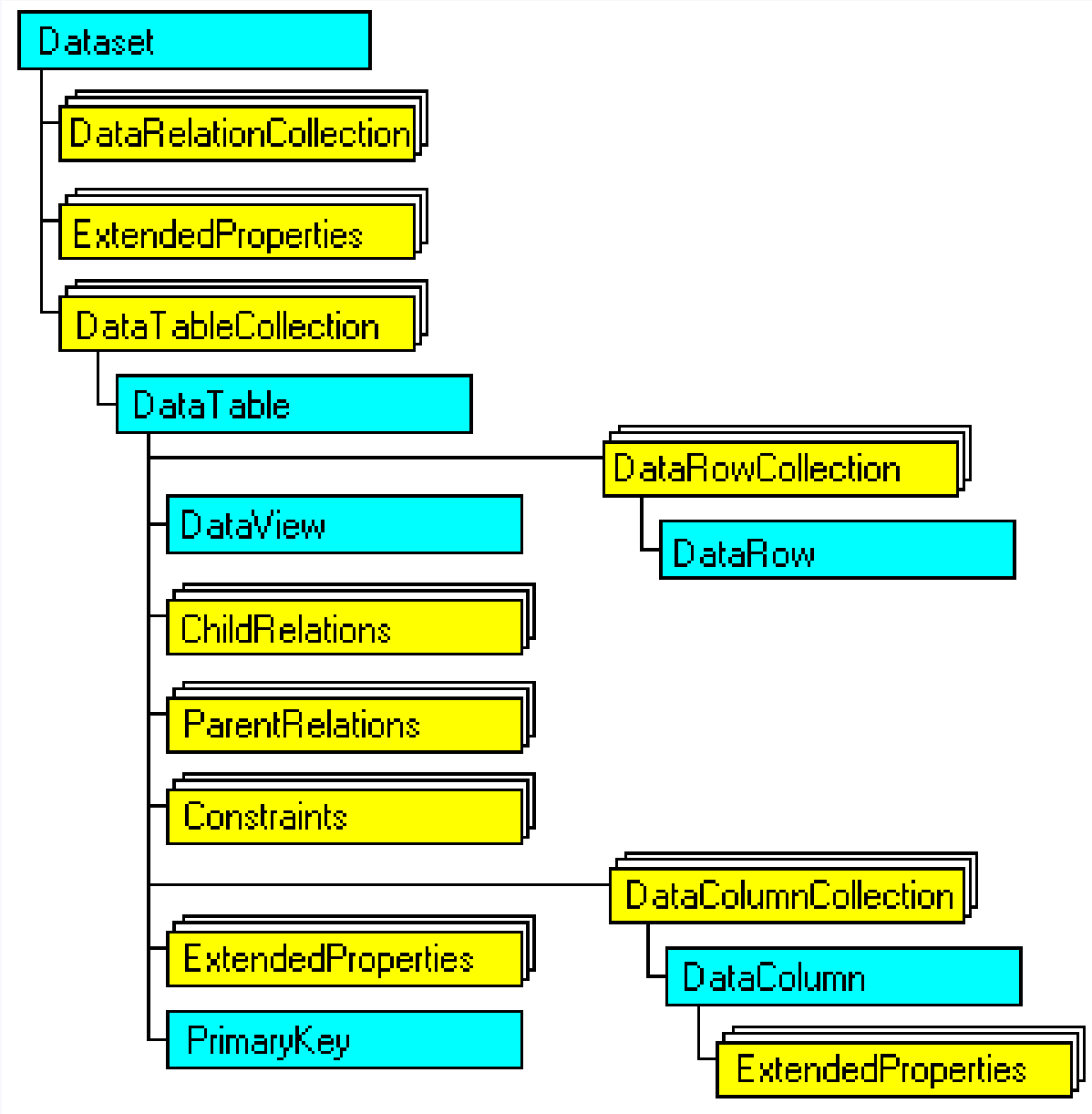
- The **DataAdapter** uses **Command** objects to execute SQL commands at the data source to both load the **DataSet** with data and reconcile changes that were made to the data in the **DataSet** back to the data source.

DataSet and DataAdapter

- For example, the SqlDataAdapter performs the following tasks when filling a DataSet with data:
  - Open connection
  - Retrieve data into DataSet
  - Close connection

- and performs the following actions when updating data source with DataSet changes:
  - Open connection
  - Write changes from DataSet to data source
  - Close connection

DataSet


Dataset
DataRelationCollection
ExtendedProperties
DataTableCollection
DataTable
DataRowCollection
DataRow
DataView
ChildRelations
ParentRelations
Constraints
DataColumnCollection
DataColumn
ExtendedProperties
ExtendedProperties
PrimaryKey

DataAdapter

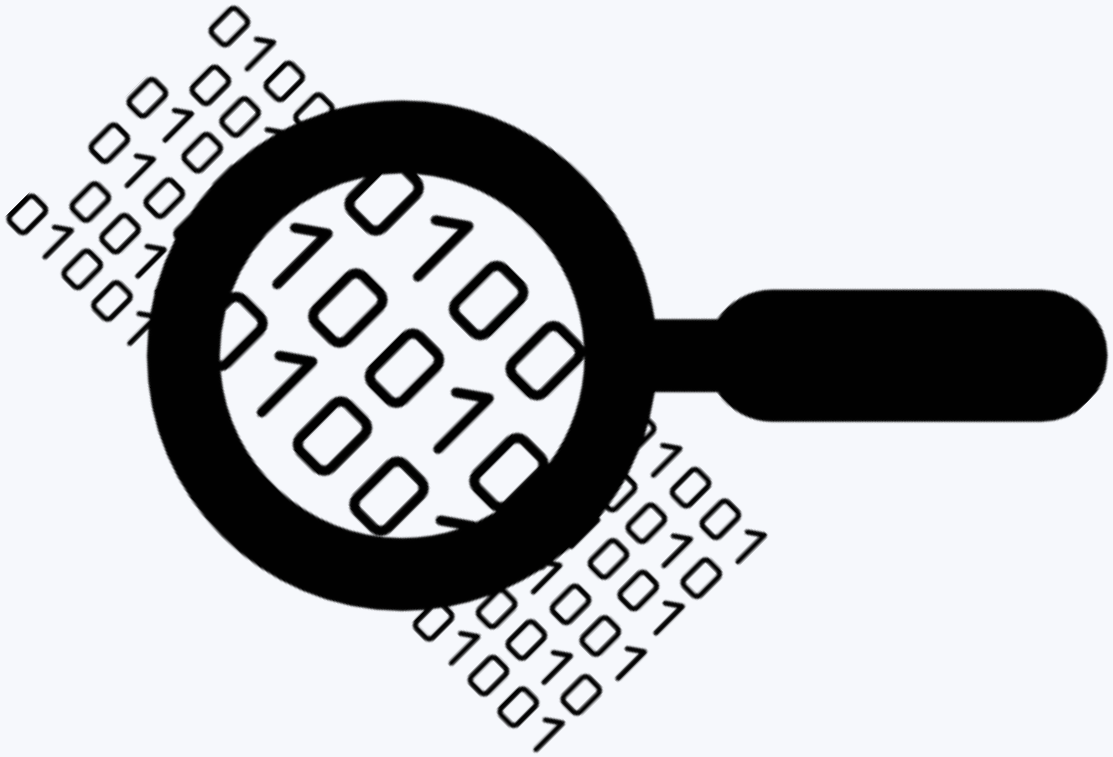
```
_dbDataAdapter = new SQLiteDataAdapter("SELECT * FROM Participant", _conn);

//delete
var deleteCommand = new SQLiteCommand("DELETE FROM Participant WHERE Id = @Id", _ conn);
deleteCommand.Parameters.Add(new SQLiteParameter("@Id"));

_dbDataAdapter.DeleteCommand = deleteCommand;

//insert
var insertCommand = new SQLiteCommand("INSERT INTO Participant (LastName, FirstName, BirthDate)
VALUES (@LastName, @FirstName, @BirthDate);", _ conn);
insertCommand.Parameters.Add(new SQLiteParameter("@LastName"));
…….. //FirstName, BirthDate

_dbDataAdapter.InsertCommand = insertCommand;
```

Choosing a DataReader or a DataSet

- Use a **DataSet** to do the following:
  - Cache data locally in your application so that you can manipulate it. If you only need to read the results of a query, the **DataReader** is the better choice.
  - Interact with data dynamically such as binding to a Windows Forms control or combining and relating data from multiple sources.
  - Perform extensive processing on data without requiring an open connection to the data source, which frees the connection to be used by other clients.

- If you do not require the functionality provided by the **DataSet**, you can improve the performance of your application by using the **DataReader** to return your data in a forward-only, read-only manner.

Demo



Custom Controls

Custom Controls

- With the .NET Framework, you can develop and implement new controls, that can be used in multiple places within an application or organization.

- Common scenarios:

Extended Controls

- Extending an existing control to customize it or to add to its functionality.
- An e-mail address TextBox, a numeric TextBox (include validations)
- A button whose color cannot be changed and a button that has an additional property that tracks how many times it has been clicked are examples of extended controls.
- You can customize any Windows Forms control by deriving from it and overriding or adding properties, methods, and events.

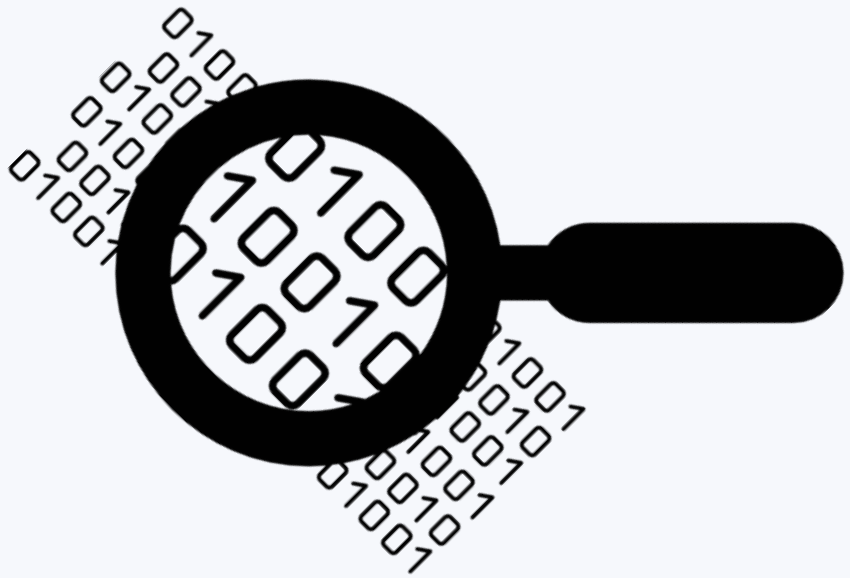
Composite Controls

- Composite controls encapsulate a user interface that can be reused as a control.
- An example of a composite control is a control that consists of a text box and a reset button.
- To author a composite control, derive from System.Windows.Forms.UserControl. The base class UserControl provides keyboard routing for child controls and enables child controls to work as a group.
- More information: Developing a Composite Windows Forms Control.

Custom Controls

- Authoring a control that does not combine or extend existing controls.
- In this scenario, derive your control from the base class Control.
- You can add as well as override properties, methods, and events of the base class.
- More info: Develop a Simple Windows Forms Control.

Demo



- **Extended** Controls and **Composite** Controls:
  - NumericTextBoxUserControlSample
- **Custom** Controls:
  - BarChartGraphicsSample

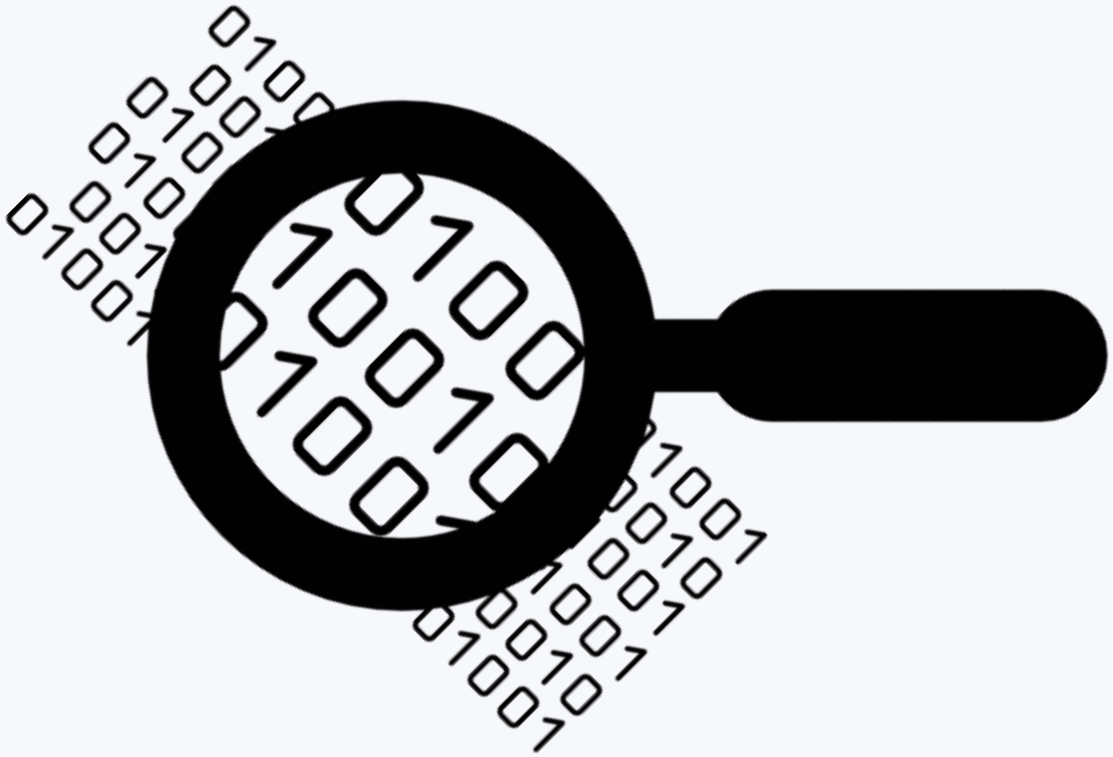
Graphics

Graphics

- The Graphics class provides methods for drawing objects to the display device. A Graphics instance is associated with a specific device context.

- You can obtain a Graphics object (the class does not include a public constructor):
  - by calling the Control.CreateGraphics method on an object that inherits from System.Windows.Forms.Control;
  - by handling a control's Control.Paint event and accessing the Graphics property of the System.Windows.Forms.PaintEventArgs class;
  - from an image by using the Graphics.FromImage method.

Demo



Clipboard

Clipboard

- Class: System.Windows.Forms.Clipboard

- The *clipboard* is a set of functions and messages that enable applications to transfer data. Because all applications have access to the clipboard, data can be easily transferred between applications or within an application.

Adding data – Multiple Formats

- Call SetDataObject to put data on the Clipboard in **one** or **multiple** formats, replacing its current contents.

SetDataObject(Object, Boolean)	Clears the Clipboard and then places data on it and specifies whether the data should remain after the application exits.
SetDataObject(Object)	Clears the Clipboard and then places nonpersistent data on it.

- For a list of predefined formats to use with the Clipboard class, see the DataFormats class.

Adding data – Multiple Formats

```
DataObject data = new DataObject();

// Add a Customer object using the type as the format.
data.SetData(new Customer("Customer as Customer object"));

// Add a ListViewItem object using a custom format name.
data.SetData("CustomFormat",  new ListViewItem("Customer as ListViewItem"));

Clipboard.SetDataObject(data);
```

Adding data – Specific Format

- To add data of a particular format to the Clipboard, replacing the existing data, call the appropriate Set*Format* method, such as SetText, or call the SetData method to specify the format.

```
//Copy HTML from web browser onto the clipboard
Clipboard.SetText(WebBrowser1.DocumentText, TextDataFormat.Html);
```

```
//Copy the picture from the picture box onto the clipboard
Clipboard.SetImage(PictureBox1.Image);
```

```
//Copy Pixel object onto the clipboard
Clipboard.SetData("Pixel", _pxl);
```

Retrieving data - Multiple Formats

- Call GetDataObject to retrieve data from the Clipboard. The data is returned as an object that implements the IDataObject interface.

- To specify the format in these operations, call the ContainsData and GetData methods instead.

```
DataObject retrievedData = (DataObject)Clipboard.GetDataObject();

if (retrievedData.GetDataPresent("CustomFormat")){
        ListViewItem item = retrievedData.GetData("CustomFormat") as ListViewItem;
        if (item != null) MessageBox.Show(item.Text);
}

if (retrievedData.GetDataPresent(typeof(Customer))){
        Customer customer = retrievedData.GetData(typeof(Customer)) as Customer;
        if (customer != null) MessageBox.Show(customer.Name);
}
```

Retrieving data - Specific Format

- To retrieve data of a particular format from the Clipboard, first call the appropriate **Contains***Format* method (such as ContainsText) method to determine whether the Clipboard contains data in that format, and then call the appropriate **Get***Format* method (such as GetText) to retrieve the data if the Clipboard contains it.

```
if (Clipboard.ContainsText(TextDataFormat.Html))
        TextBox2.Text = Clipboard.GetText(TextDataFormat.Html);
```

```
if (Clipboard.ContainsImage())
        PictureBox2.Image = Clipboard.GetImage();
```

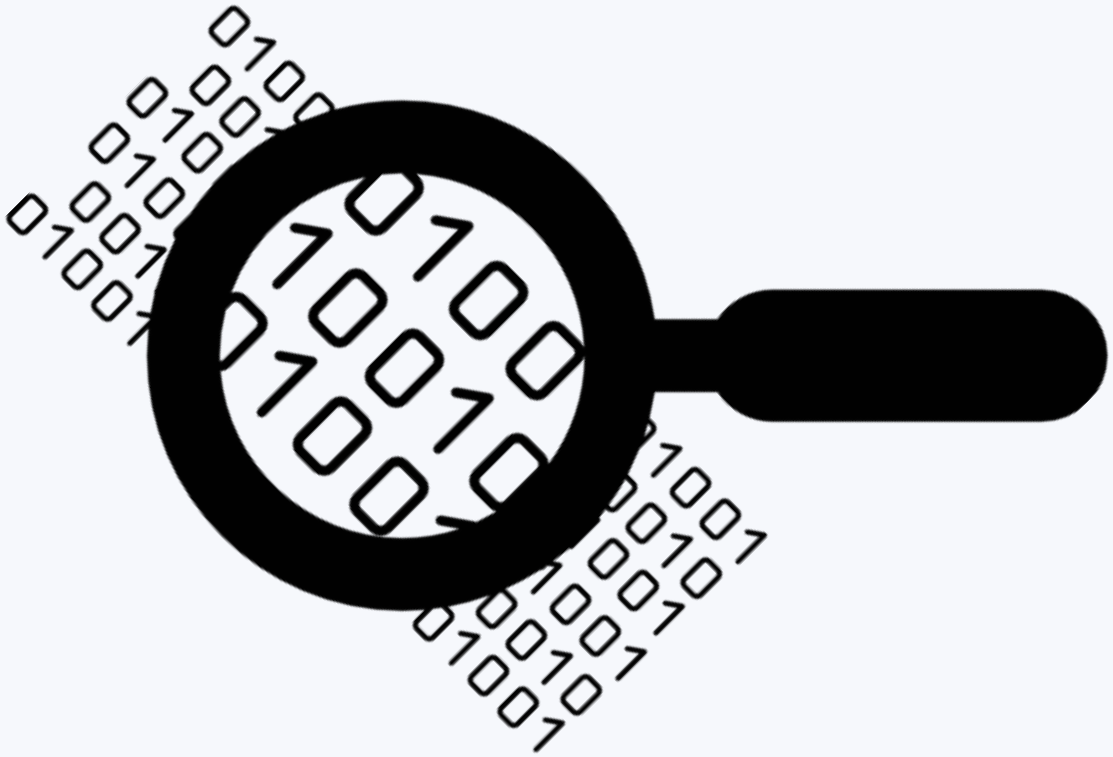
```
if (Clipboard.ContainsData("Pixel")){
    var tempPixel = (Pixel) Clipboard.GetData("Pixel");
    LoadPixel(tempPixel, TextBox4);
}
```

Clear

- Call the Clear method to remove all data from the Clipboard.

```
//Clear the clipboard
Clipboard.Clear();
```

Demo



Drag & Drop

Performing

- In order to begin a drag-and-drop operation, the **DoDragDrop** method should be called.

```
public DragDropEffects DoDragDrop(
        object data,
        DragDropEffects allowedEffects
)
```

- Parameters
  - **data:** the data to drag (a base managed class such as String or Bitmap, or an object that implements ISerializable or IDataObject.)
  - **allowedEffects:** Copy, Move

Performing

- Example:

```
private void tbText_MouseDown(object sender, MouseEventArgs e)
{
        // This initiates a DragDrop operation, specifying that the data to be dragged
        // with be the text stored in the tbText control.
        tbText.DoDragDrop(tbText.Text, DragDropEffects.Copy);
}
```

Control.AllowDrop Property

- Gets or sets a value indicating whether the control can accept data that the user drags onto it.

- **true** if drag-and-drop operations are allowed in the control; otherwise, **false**. The default is **false**.

Events

Mouse Event	Description
DragEnter	This event occurs when an object is dragged into the control's bounds. The handler for this event receives an argument of type DragEventArgs.
DragOver	This event occurs when an object is dragged while the mouse pointer is within the control's bounds. The handler for this event receives an argument of type DragEventArgs.
DragDrop	This event occurs when a drag-and-drop operation is completed. The handler for this event receives an argument of type DragEventArgs.
DragLeave	This event occurs when an object is dragged out of the control's bounds. The handler for this event receives an argument of type EventArgs.

Events - DragEnter

- Example:

```
private void listBox_DragEnter(object sender, DragEventArgs e){

        // This control will use any dropped data to add items to the listbox.  Therefore,
only data in a text format will be allowed.  Setting the  autoConvert parameter to true
specifies that any data that can be converted to a text format is also acceptable.
        if (e.Data.GetDataPresent(DataFormats.Text, true)) {
                // Show the standard Copy icon.
                e.Effect = DragDropEffects.Copy;
        }
}
```

Events - DragDrop

- Example

```
private void listBox_DragDrop(object sender, DragEventArgs e)
{
        if (e.Data.GetDataPresent(DataFormats.Text, true))
        {
                // Create the list item using the data provided by the source control.
                listBox.Items.Add(e.Data.GetData(DataFormats.Text));
        }
}
```

Drag & Drop

Further reading: docs.microsoft.com/en-us/dotnet/desktop/winforms/drag-and-drop-functionality-in-windows-forms

Printing

Steps

1. Crate an instance of the **PrintDocument** class

2. (Optional) Set properties such as the **DocumentName** and **PrinterSettings**

3. Call the **Print** method to start the printing process.

4. Handle the **PrintPage** event in order to specify the output to print, by using the GraphicsGraphics property of the PrintPageEventArgs.

Steps

1. Crate an instance of the **PrintDocument** class

2. (Optional) Set properties such as the **DocumentName** and **PrinterSettings**

3. Call the **Print** method to start the printing process.

4. Handle the **PrintPage** event in order to specify the output to print, by using the GraphicsGraphics property of the PrintPageEventArgs.

Printing

Further reading: link

Releasing the App

Releasing the App

Deployment using Click Once

Publish to Microsoft Store

Deployment with ClickOnce

- **ClickOnce** is a deployment technology that enables you to create self-updating Windows-based applications that can be installed and run with minimal user interaction.

- Docs: docs.microsoft.com/en-us/visualstudio/deployment/publishing-clickonce-applications

Creating a Setup

- Publish the application using the wizard
- Configure settings in the project Properties

Publish to Microsoft Store

- Docs: docs.microsoft.com/en-us/windows/apps/desktop/modernize/desktop-to-uwp-distribute