

Creating Managed Smart Client Applications

Software techniques



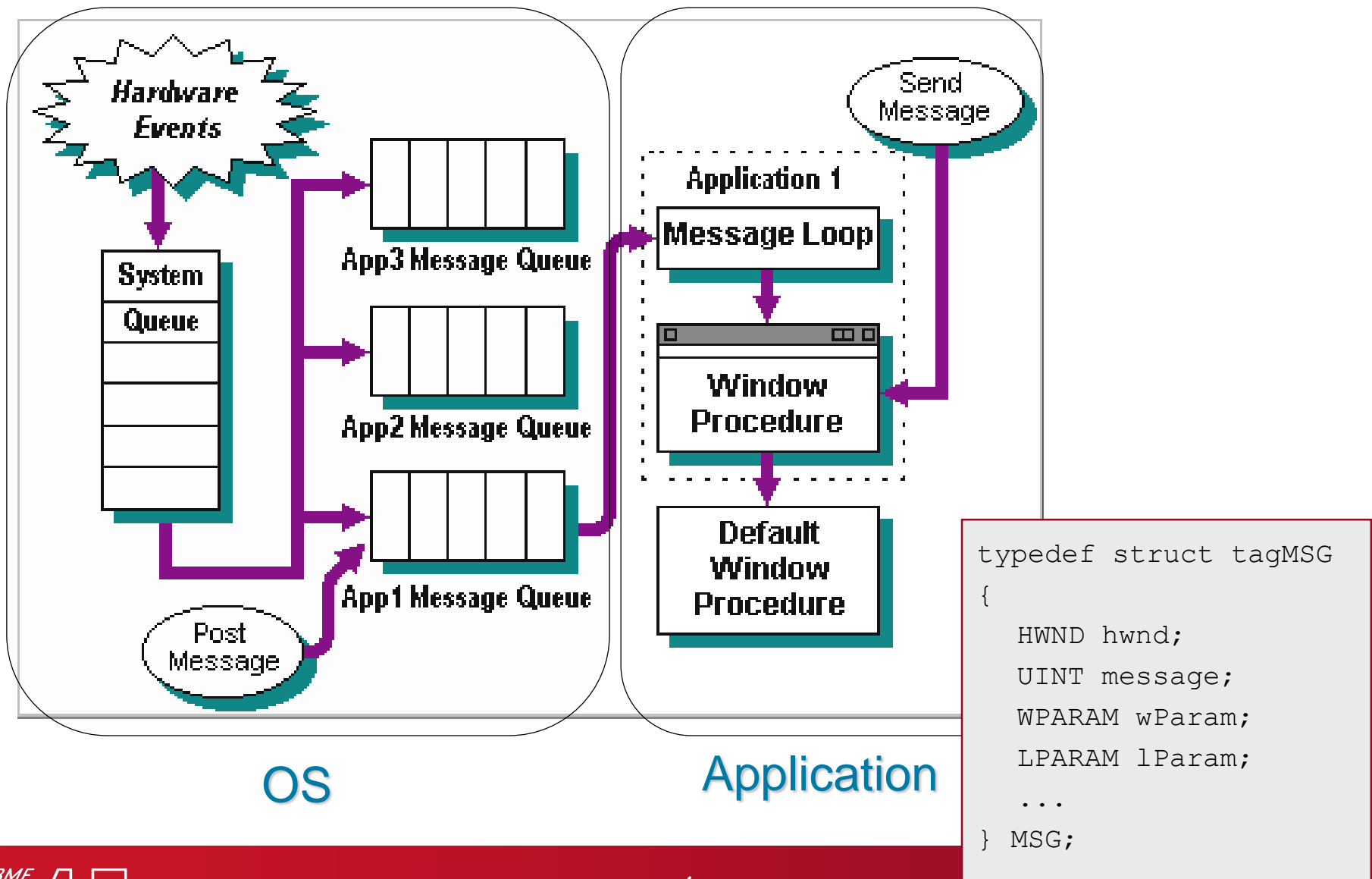
Department of
Automation and
Applied Informatics

Content

- Native event-driven platform (Win32)
- .NET Windows Forms Applications
 - > The connection between the native and the managed platform
 - > Architecture
 - > .Net partial classes
 - > Events
 - > Dialog windows
 - > Menu and toolbar
 - > Control hierarchy
 - > Destructors and the dispose
 - > Creating custom controls

NATIVE EVENT-DRIVEN PLATFORM (WIN32)

Event handling with the Win32 platform

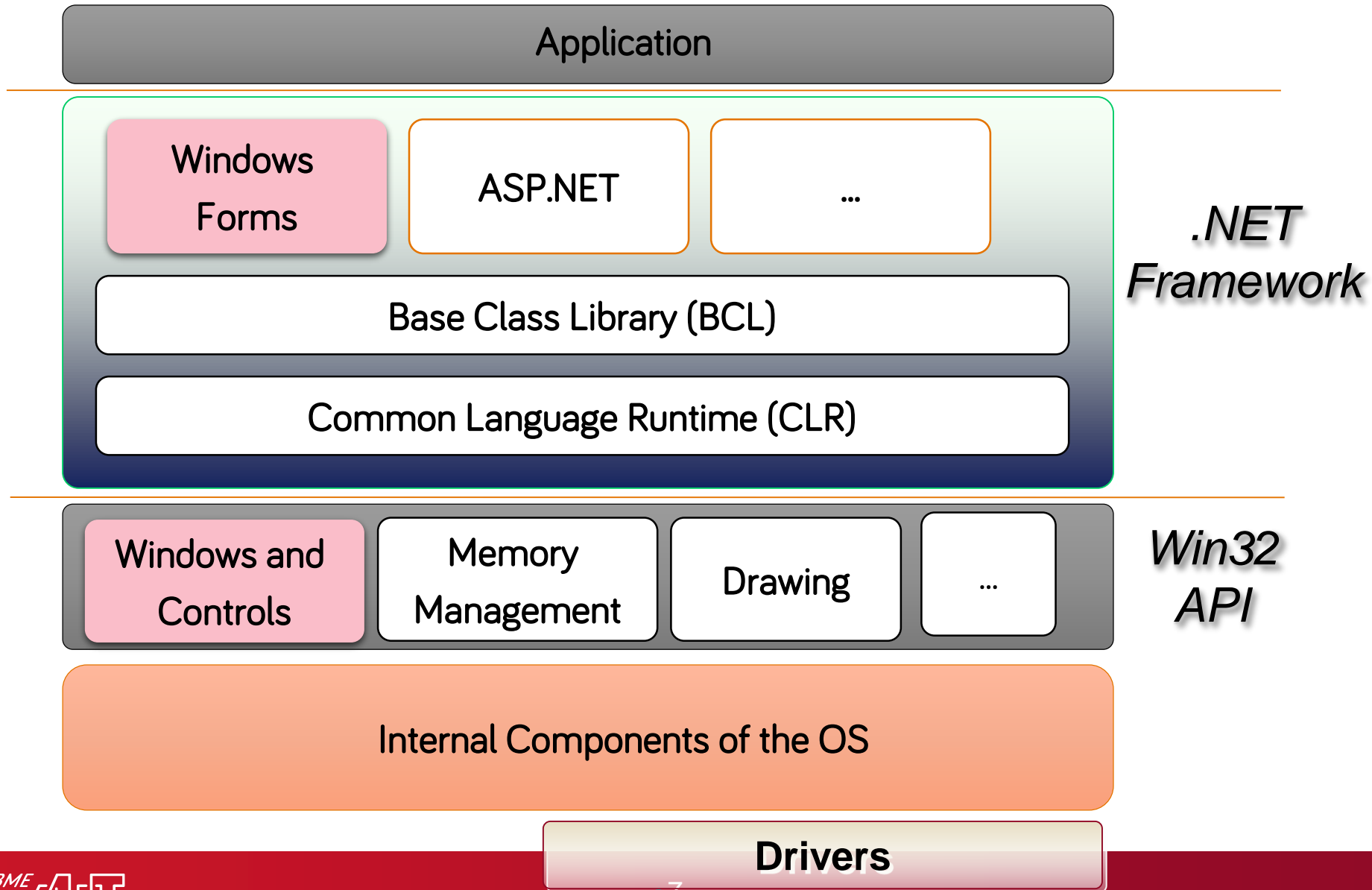


Win32 API vs .NET Windows Forms

- Win32 API window handling
 - > Native, only the OS is needed
 - > Basically C and C++ (theoretically any language can be used that supports the API)
 - > Hard to create a nice structure (switch-case structure for handling the different messages)
 - > Not object oriented
 - > Hard to work with complex controls (tree, list, tab)
- Windows Forms
 - > .NET
 - > Less code → more efficient development
 - > More facilities

THE ARCHITECTURE OF WINDOWS FORMS APPLICATIONS

Technology stack



Partial classes

- Only from .NET 2.0
- The compiler merges the two files (the files cannot be in separate assemblies)
- Main usage area: separating the generated and the hand-written code

```
public partial class Customer
{
    private int id;
    private string name;
    private List<Orders> orders
}
```

```
public partial class Customer
{
    public void SubmitOrder(Ord
        orders.Add(order);
    }

    public bool HasOutstandingO
        return orders.Count > 0;
    }
}
```

```
public class Customer
{
    private int id;
    private string name;
    private List<Orders> orders;

    public void SubmitOrder(Order order) {
        orders.Add(order);
    }

    public bool HasOutstandingOrders() {
        return orders.Count > 0;
    }
}
```


The Forms class

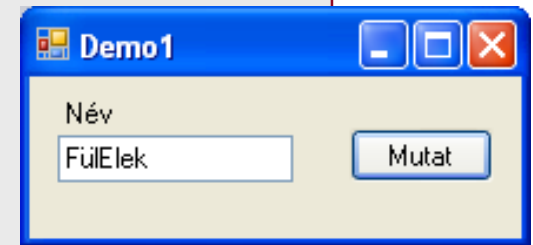
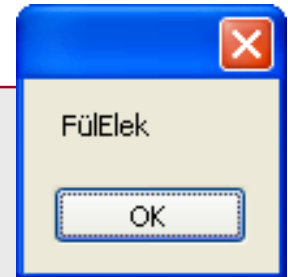


- System.Windows.Forms namespace
- Every window is a class that derives from the Form class
- The Form class
 - > Has got many properties (e.g. BackColor, Text, Size, ...)
 - > Publishes many events (Load – when loaded, Click – mouse click, Resize – when resized, Move – when relocated, KeyDown – when a key is pressed, ...)
- We can place controls onto a Form (e.g. TextBox, Label, etc.)
 - > Either by the Visual Studio designer, from the ToolBox
 - > Or programmatically
- The controls
 - > The controls will be the member variables of the form object
 - > They are instantiated in the InitializeComponent() method that is called by the constructor
 - > Has got many properties (e.g. Font property) and publishes many events (e.g. TextBox publishes the TextChanged event)
- Visual Studio generates partial classes

Demo

- If we click on the button it will display the content of the textbox in a message box

```
partial class MainForm
{
    private System.Windows.Forms.TextBox tbName;
    private System.Windows.Forms.Label Label1;
    private System.Windows.Forms.Button buttonShowName;
    ...
    private void InitializeComponent()
    {
        this.tbName = new System.Windows.Forms.TextBox();
        this.tbName.Location = new System.Drawing.Point(12, 25);
        this.tbName.Size = new System.Drawing.Size(100, 20);
        ...
        this.buttonShowName = new System.Windows.Forms.Button();
        this.buttonShowName.Location = new System.Drawing.Point(136, 22);
        this.buttonShowName.Size = new System.Drawing.Size(61, 23);
        this.buttonShowName.Click +=
            new System.EventHandler(this.buttonShowName_Click);
    }
}
```



Demo part 2

- The main window is the parameter of the `Application.Run`:

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

    private void buttonShowName_Click(object sender, EventArgs e)
    {
        MessageBox.Show(tbName.Text);
    }
}
```

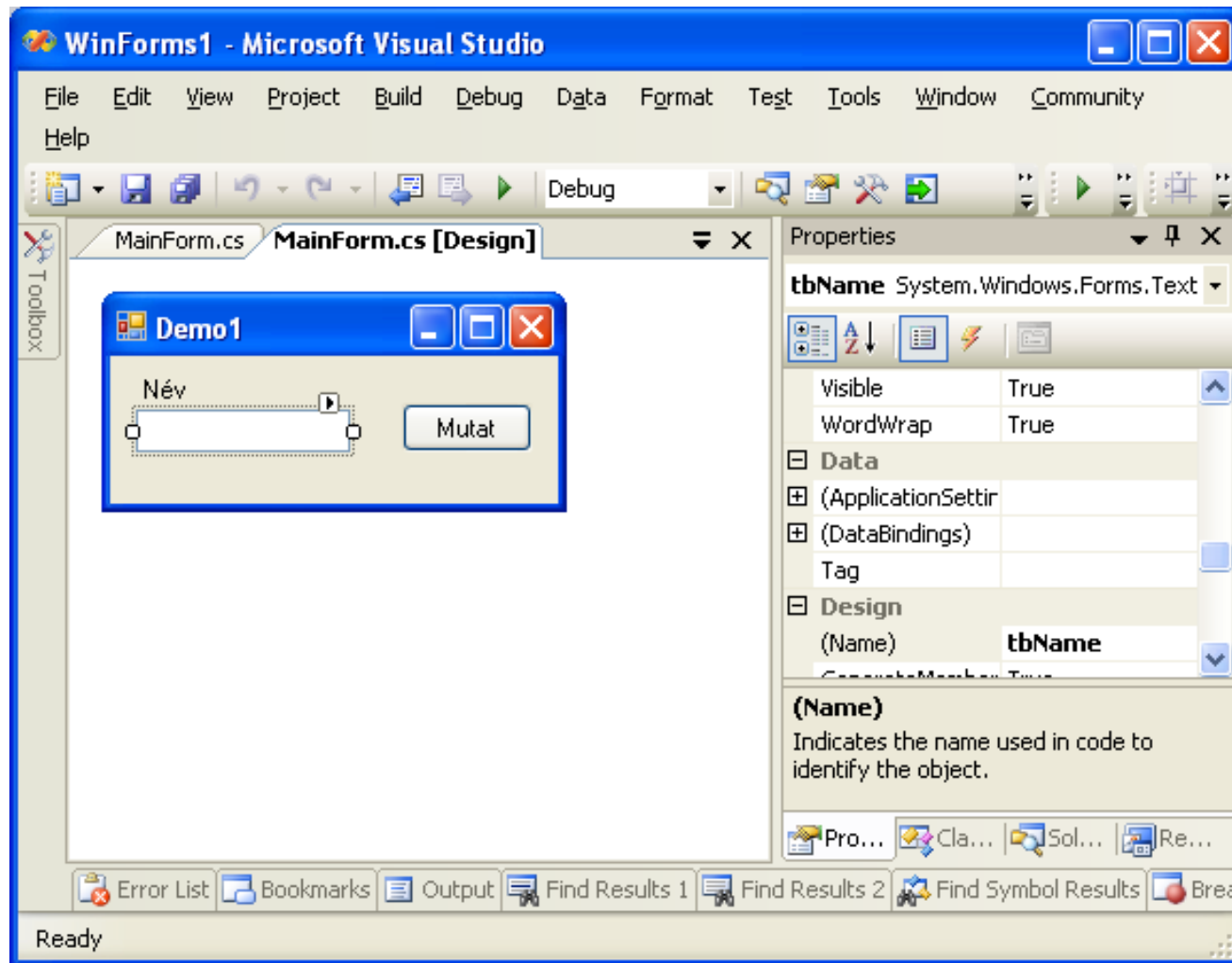
```
static class Program
{
    static void Main()
    {
        ...
        Application.Run(new MainForm());
    }
}
```

Event-driven programming

Event-driven programming

- We design the user interface
- The execution of the program is controlled by the user events and how we react to them (e.g. pressing a button, selecting a menu item, clicking with the mouse). There are system events as well (e.g. timer).
- The events of the controls on a form are handled inside the form class

The designer of the Visual Studio



Message and event handling

- .NET applications build on the native message and window handling
 - > The forms and the controls has got handles (HWND, Control.Handle property)
 - > The form is a wrapper around a native window
 - > The form and the controls convert the native messages to .NET events
 - > An application (GUI thread) has got a message loop
 - Prove it!
 - Executing it manually: Application.DoEvents(): this will process the messages in the message queue.
 - > Starting the application:

```
static void Main()
{
    ...
    // The main window is the parameter:
    // the app is closes when the window is closed.
    Application.Run(new MainForm());
}
```

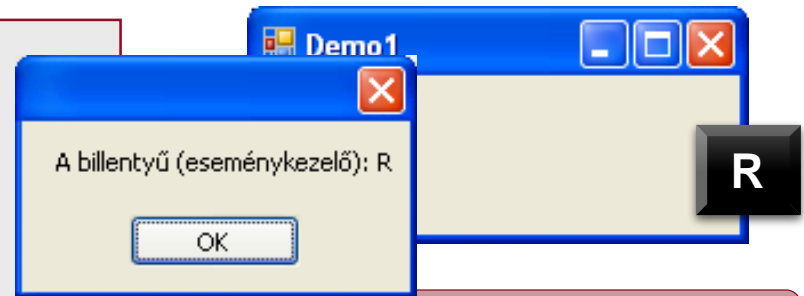
Message and event handling

```
public partial class MainForm : Form
{
    public MainForm()
    {
        InitializeComponent();
        this.KeyDown +=
            new KeyEventHandler(this.MainForm_KeyDown);
    }

    protected override void OnKeyDown(KeyEventArgs e)
    {
        base.OnKeyDown(e);
        MessageBox.Show(„The key (virt. func.): "
            + e.KeyCode.ToString());
    }

    private void MainForm_KeyDown(object sender,
        KeyEventArgs e)
    {
        MessageBox.Show(„The key (event handler): "
            + e.KeyCode.ToString());
    }

    ...
}
```



Pressing the 'R' key

WM_KEYDOWN into the
message queue of the app

The hidden message-loop
removes it from the queue

The window/form gets the
message (WndProc)

virtual void OnKeyDown (
[KeyEventArgs](#) e) is called

event [KeyEventHandler](#)
KeyDown is fired

MenuStrip, toolstrip and statusstrip

- MenuStrip

- > Elements: ToolStripMenuItem, ToolStripTextBox, ToolStripComboBox, separator

- ToolStrip

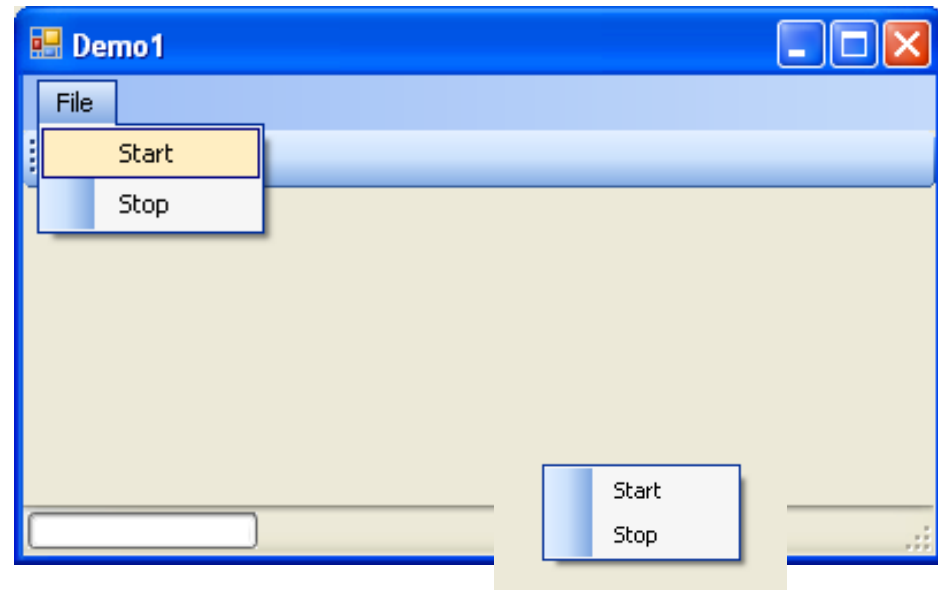
- > Elements: ToolStripButton, etc.

- StatusStrip

- > Elements: ToolStripStatusLabel, etc.

- Context menu

- > ContextMenuStrip



Timers

- `System.Windows.Forms.Timer` component
 - > Periodically fires an event
 - > Interval property: the time interval in milisec
 - > bool Enabled property: enables or disables the timer
 - > Start() and Stop() operations
 - > Tick event (EventHandler type), the event
 - > Inaccurate (the minimal granularity is about 20 ms)
- `System.Threading.Timer`
 - > Very simple
 - > Calls a callback (delegate) in a ThreadPool thread
 - > Used in multi-threaded server applications (don't use them with Forms, if possible)
 - > Accurate
- `System.Timers.Timer`
 - > Similar to the `System.Threading.Timer`, but with more facilities

Dialog windows

- The purpose of dialog windows
 - > Usually for displaying and modifying some sort of settings
 - > Modal display (we cannot switch to an other window)
- To create a new form: right click on the project + Add/New Form
- Border style for the form
 - > FormBorderStyle property
 - FormBorderStyle.Sizeable (def.) - resizable
 - **FormBorderStyle.FixedDialog** – non-resizable
 - ...
- Dialog window
 - > Create properties for the settings in our Form class
 - Interval property in the example
 - > The Form uses the DialogResult property to express if the new values are valid (if the window was closed by pressing the OK button). Possible values:
 - DialogResult.Ok, DialogResult.Cancel, DialogResult.Yes, ...

Using dialog windows

```
public partial class SettingsForm : Form
{
    private int interval;
    public int Interval
    {
        get { return interval; }
        set
        {
            interval = value;
            textBox1.Text = value.ToString();
        }
    }

    private void bOk_Click(object sender, EventArgs e)
    {
        if (!int.TryParse(textBox1.Text, out interval))
            MessageBox.Show(„Invalid value!");
        else
            this.DialogResult = DialogResult.OK;
    }
}
```

Displaying dialog windows

```
private void settingsToolStripMenuItem_Click(object sender, EventArgs e)
{
    SettingsForm form = new SettingsForm();
    form.Interval = timer1.Interval;
    if (form.ShowDialog() == DialogResult.OK)
    {
        MessageBox.Show("Changed!");
        timer1.Interval = form.Interval;
    }
}
```

- Specify the initial values for the settings
- Show the form in a modal way: `Form.ShowDialog()`
- Test its result: if it was closed by the Ok button (`DialogResult` enum type)

MessageBox

```
string message = "You did not enter a server name. Cancel this operation?";  
string caption = "No Server Name Specified";  
MessageBoxButtons buttons = MessageBoxButtons.YesNo;  
DialogResult result;  
  
result = MessageBox.Show(this, message, caption,  
    buttons, MessageBoxIcon.Question);  
if(result == DialogResult.Yes)  
{  
    this.Close();  
}
```



Non-modal windows

- An other window can also be activated
- Displaying

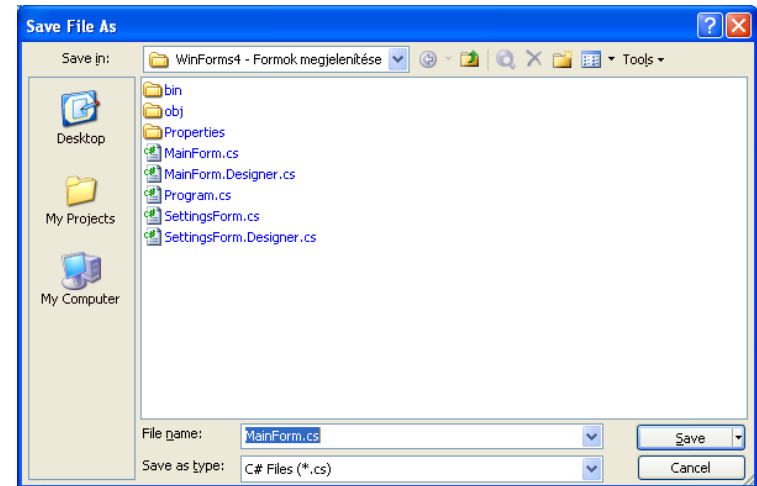
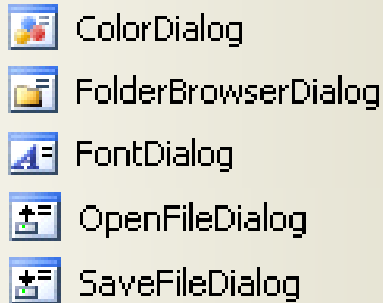
```
SettingsForm form = new SettingsForm();  
form.Show();
```

- In the Show() we can specify an owner window
 - > Can not be behind the owner windows (Z-order)

```
SettingsForm form = new SettingsForm();  
form.Show(this); // The 'this' is a class that derives from Fo.
```

Common dialogs

- .NET wrappers around the Win32 API common dialogs



+PageSetupDialog, PrintDialog, PrintPreviewDialog

```
OpenFileDialog ofd = new OpenFileDialog();  
ofd.InitialDirectory = @"c:\";  
ofd.Multiselect = false;  
ofd.Filter = "C# Files (*.cs)|*.cs|All files (*.*)|*.*";  
  
if ( ofd.ShowDialog() == DialogResult.OK )  
{  
    MessageBox.Show(„The path of the file: " + ofd.FileName );  
}
```

Important simple controls

 MonthCalendar

 NotifyIcon

 NumericUpDown


 PictureBox

 ProgressBar

 RadioButton

 RichTextBox

 TextBox

 ToolTip

 TreeView

 WebBrowser

 DataGridView

 Button

 CheckBox

 CheckedListBox

 ComboBox

 DateTimePicker

 Label

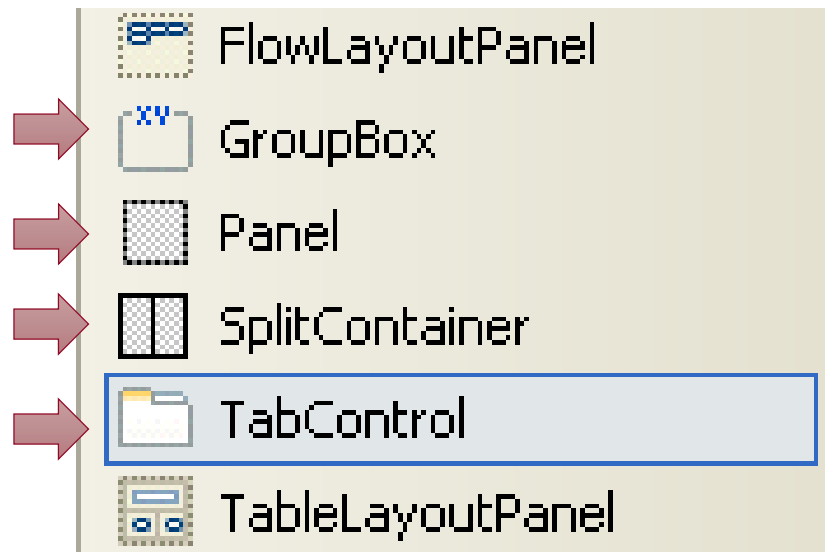
 LinkLabel

 ListBox

 ListView

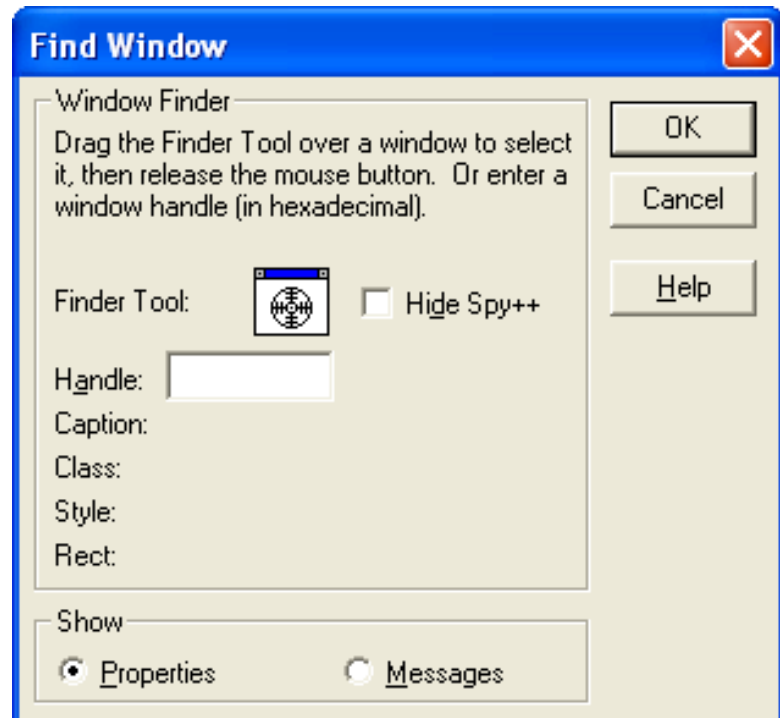
 MaskedTextBox

Container controls

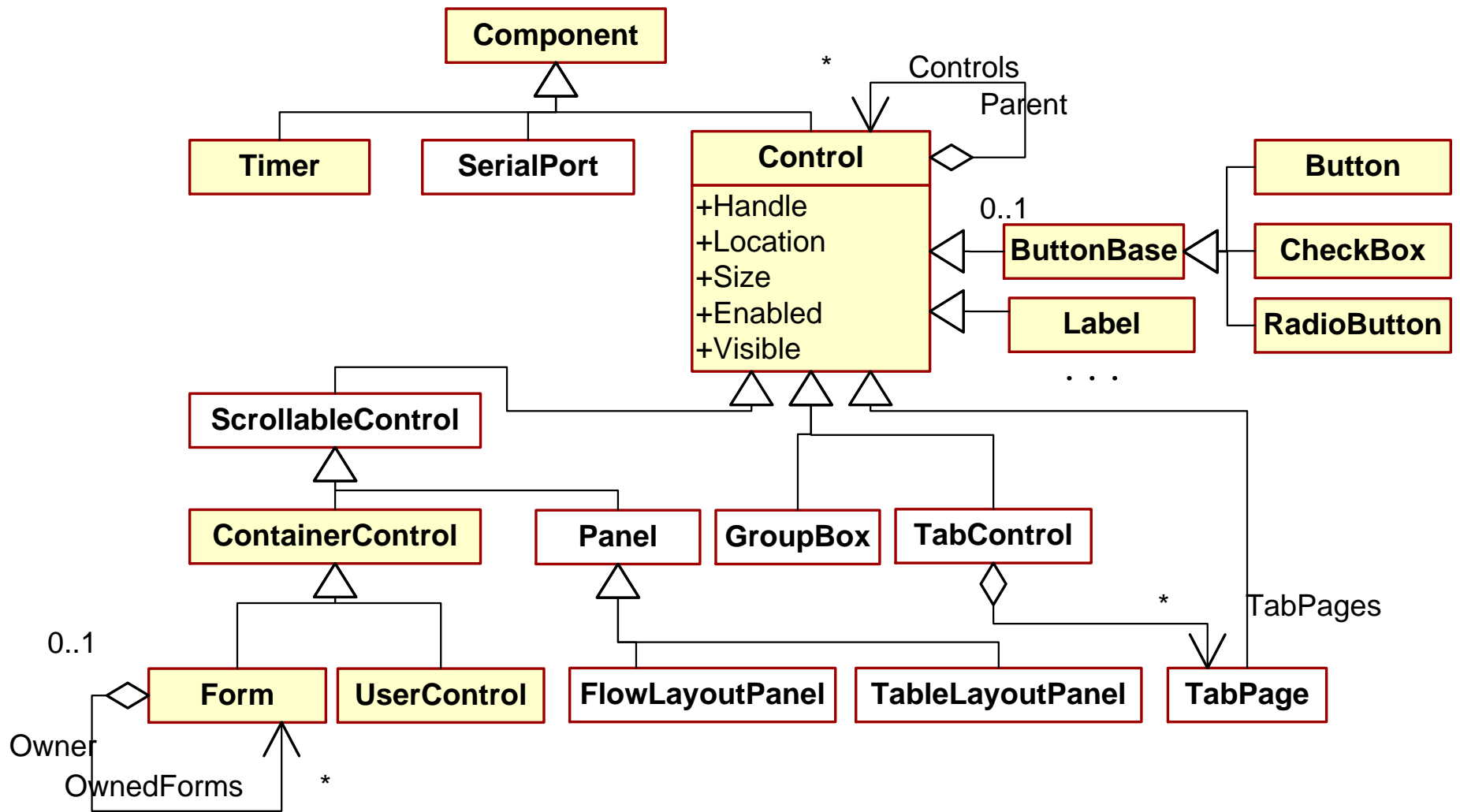


- For the grouping of simple or other complex controls
- TabPages can be added to TabControls

E.g. GroupBox:



Component/Control hierarchy



Control hierarchy

- Component:

- > Can be contained by any *container* (e.g. *designer*)
- > No need to have graphical interface (e.g. *SerialPort*), but it can be used in the designer (VS) and its properties and events can graphically be edited

- Control:

- > The base class for all controls
- > Has got a native window handle (Handle property)
- > Common properties for the controls
 - **Visible**
 - **Enabled**
 - **Controls** – The list of contained controls (only for **ContainerControls**)
 - Location, Size, ...
- > Common methods for the controls
 - **Focus()**, ...
- > Common events for controls
 - **Click**
 - **KeyDown**, ...

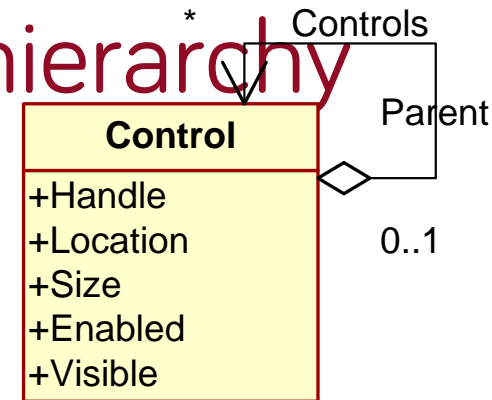
Control hierarchy

- ContainerControl
 - > Container for other controls
 - > Responsible for focus management. If an other *ContainerControl* gets the focus it still stores its active control (ActiveControl property)

Controls and forms – ‘part of’ hierarchy

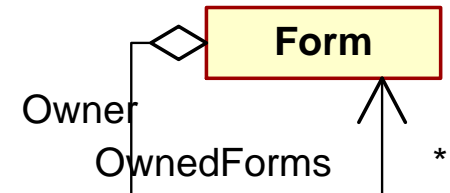
- Parent – child relationship

- > Control.Controls property
 - The contained controls are child windows
- > Guarantees the expected behavior
 - The child windows move with the parent window
 - If the parent window dies the child windows will die as well
 - The child windows cannot be outside of the parent window (clipping)
 - Hiding/showing the parent window automatically hides/shows all the child windows
 - Parent property to access the parent window (null, when there is no)



- Between forms: owner – owned relationship

- > Looser than parent-child
 - If the owner is closed all the owned will be closed (+ the same for minimize)
 - The owned is always in front of the owner (Z-order)
 - Owned forms: **Form.OwnedForms** property
 - Owner form: **Form.Owner** property (null, if there is no)



Important events

- The type of the delegate is usually (if there is no special event param):

```
public delegate void EventHandler ( Object sender, EventArgs e )
```

- > A sender: the publisher of the event
 - > EventArgs: the parameter of the event
 - The EventArgs contains no information
 - We must derive from EventArgs if there are params for the event
- Key events (Control events)
 - > →

Key events

Event	Description	Delegate	Event parameters
KeyDown	Key is pressed down	void KeyEventHandler (Object sender, EventArgs e)	EventArgs.Shift – the Shift is pressed (the same for Alt and Control) EventArgs.KeyCode – the key without the modifiers (Keys enum type) EventArgs.KeyData - the key with the modifiers (Keys enum type)
KeyUp	Key is released	Mimic a KeyDown	Just like KeyDown
KeyPress	Repeating if the key is pressed and held	void KeyPressEventHandler (Object sender, KeyPressEventArgs e)	KeyPressEventArgs.KeyChar – the ASCII code for the pressed key, char type

```
this.KeyDown += new KeyEventHandler(this.MainForm_KeyDown);  
...  
private void MainForm_KeyDown(object sender, EventArgs e)  
{  
    // test if ALT + E was pressed together  
    if(e.Alt && e.KeyCode == Keys.E) { ... }
```

Layout management

- Lab...
 - > Anchor
 - > Dock
 - > Splitter - SplitContainer

Destructors, Finalize and the Dispose pattern

Destructors in C++

- No garbage collection, the programmer manages memory
- Usually it's a class' responsibility to free any dynamic memory it allocates

```
class Stack {  
private:  
    int* pFirst; int* pCurrent; int size;  
public:  
    void push(int n) { ... } ;  
    int pop() { ... };  
    Stack(int size): size(size)  
    {  
        pCurrent = pFirst = new int[size];  
    }  
    ~Stack() { delete[] pFirst; }  
};
```

.NET - Finalize and destructors

- There is no free/delete, objects that are no longer referenced are freed by the garbage collector - GC

```
public class Stack
{
    int[] items;
    public Stack(int size)
    {
        items = new int[size];
    }
    ...
    ~Stack()
    {
        delete items; // DOES NOT EXIST!
    }
}
```

- If no other object refers to a Stack object, then no object can refer to int [] items: the GC can free this too.
 - > So we don't need a destructor.... - ever?

.NET - Finalize and destructors

- When the GC frees an object, its Finalize method is called (CLR term)
 - > This is what we call the Finalizer
 - > However, in C# we do not override the Finalize methods ourselves. We write a destructor (~class_name {...})! This code:

```
class MyClass{  
    ~MyClass() { // Clean up code... }  
}
```

- > will be implicitly translated to this:

```
class MyClass {  
    protected override void Finalize()  
    {  
        try  
        { // Clean up code... }  
        finally { base.Finalize(); }  
    }  
}
```

Key destructor/Finalizer features



- Finalizer/destructor features
 - > We can't call it directly and we don't know when it will be called! (The CLR invokes it when it runs the GC-t)
 - > We don't know the order in which the finalizers are called
 - If we have 100 objects, we don't know the order in which they will be freed (they might reference eachother!)
 - > We don't know what thread will run them!
- Note
 - > Only write a destructor if you have to. All objects with destructors (even with empty body) go into the finalizer queue: this slows things down unnecessarily!

Key destructor/Finalizer features

- Is this ok?

```
public class Stack
{
    int[] items;
    public Stack(int size)
    {
        items = new int[size];
    }
    ...
    ~Stack()
    {
        Console.WriteLine(items.Length);
    }
}
```

- We don't know when items is collected. If it happens faster than the rest of the Stack object, we reference an object that no longer exists. Be careful! Don't free dynamically allocated memory yourself in the destructor.
- So, why would we ever write a destructor??? →

When we DO write a destructor

```
public class MySuperFileReader
{
    // Native file handle - a non-managed resource
    private IntPtr fileHandle;

    public MySuperFileReader(String fileName) {
        // We call a function from a native DLL,
        // this opens the file.
        fileHandle = FileFunctions.CreateFile(fileName,...);
        ...
    }

    ~MySuperFileReader() {
        // We call a function from a native DLL,
        // this closes the file.
        FileFunctions.CloseHandle(fileHandle);
    }
}
```

When we DO write a destructor

- „MySuperFileReader” uses a NON-MANAGED resource (in this case a file handle). These are not collected by the GC, it's our responsibility to manage them. We free it in the destructor.
- What do we want to achieve?
 - > We don't want to leak non-managed resources, we make sure to free them. This is why we write destructors.
- Some non-managed resources:
 - > files
 - > Native locks, mutexes
 - > Database connections
 - > All native windows
 - > Everything that we receive a pointer/handle/reference to from the OS.

Finalizer and destructor: summary

- When do we write a destructor
 - > There is no need to free up memory
 - > Only when the object uses an „expensive“, *unmanaged resource* (e.g. native window, database connection, file handle, lock, etc.) .
 - > Otherwise don't write, as classes that has got a destructor will get into the finalizer queue: makes it slower!
- Particularities of the finalize/destructor
 - > We don't know when it will be executed!
 - > We don't know the order of the execution!
 - > We don't know the thread that will execute it!
 - > The destructor just needs to release the unmanaged resources as the GC might have deleted the managed references
 - DON'T FREE UP MANAGED MEMBER VARIABLES IN THE DESTRUCTOR!

Dispose

- Force the GC to run
 - > Don't use it, trust the CLR!
- ```
System.GC.Collect();
```
- What to do if we have got an expensive unmanaged resource, and the execution of the finalizer is non-deterministic?
    - > We can't call it directly, so we use the Dispose pattern!
  - To have a class implement the Dispose pattern
    - > Implement the IDisposable interface!
      - Write the void Dispose() operation!
      - Inside of it release the unmanaged resources!
      - Dispose can be called explicitly, if we don't need the resource call Dispose() on it -> THAT WAS THE GOAL
      - Call the Dispose() from the destructor as well to release the unmanaged resources
  - Exact solution: „dispose pattern.doc”
  - In many cases instead of Dispose() we can call Close() and that will call Dispose(), e.g. File.Close() calls Dispose()

# Using classes that implement the IDisposable interface

- Dispose() has to be called even when there was an exception!
- Solution 1: try-finally

```
ResourceWrapper r1 = new ResourceWrapper();
try
{
 // using object r1
 r1.DoSomething();
}
finally
{
 // check if it is null
 if (r1 != null) r1.Dispose();
}
```

- Solution 2: using keyword →

# The using expression

- Or this way:

```
using (ResourceWrapper r1 = new ResourceWrapper())
{
 // use the r1 object
 r1.DoSomething();
}
```

```
ResourceWrapper r1 = new ResourceWrapper();
using (r1)
{
 // use the r1 object
 r1.DoSomething();
}
```

## More than one using:

```
using (StreamReader s1 = new StreamReader(path1))
using (StreamReader s2 = new StreamReader(path2))
{
 // Az s1 és s2 StreamReader objektumok használata
}
```

# The Dispose and the controls

- The *Component* class (the base of the *Control* class) implements the *IDisposable* interface
  - > Consequences: every control implements the *IDisposable* interface
  - > A native window (has got a HWND) is an expensive resource
  - > `Form.Close()` closes the form
- If we derive from the *Form* class, it will (because it too will implement the *Dispose* pattern):
  - close the native window by calling `base.Dispose()`
  - It will call `Dispose()` on all the contained controls (*Controls* property), this will release the handle for the native windows for each control
  - It will call `Dispose()` on all components member variables that are not controls as well (e.g. *Timer*)

# Worth knowing

- **TextBox**
  - > Text property
  - > TextChanged event (EventHandler)
- **Timer**
  - > Start, Stop operations
  - > Interval property
  - > Tick event (EventHandler)
- **Label**
  - > Text property
- **Button**
  - > Click event
- **CheckBox**
  - > Checked property
  - > Click event
- **Form**
  - ◆ Text property – header
  - ◆ KeyDown, KeyUp, KeyPress events
  - ◆ **Control**
    - ◆ Visible, Enabled properties

# Summary

- Windows Form is an efficient technology for developing smart client applications
- Not perfect:
  - > Windows Presentation Foundation is a new generation technology for developing thick clients (starting from .NET 3.0)
- Topics that weren't covered
  - > Validation
  - > Inheriting Forms
  - > Creating custom controls: next class
  - > Drawing: next class
  - > Databinding: later
  - > ....

# Questions

- Introduce the message handling of the native Win32! (Message, message queue, message loop, window procedures, callback function, default window procedure)
- Introduce the .NET thick client application architecture (tires)!
- Introduce .NET partial classes. Where are they used?
- Introduce the Windows Forms application architecture!
- What is the connection between native windows messages and .NET events (e.g. pressing a key)
- How to create menus and toolstrips in Windows Forms applications?
- What is a dialog box? Write code that demonstrates their creation and their usage.



# Questions

- What is a non-modal dialog? Write code for demonstration!
- What are common dialogs?
- List the most important controls!
- List the most important container controls!
- Introduce the control hierarchy with the most important classes and their connections (Component, Control, ContainerControl)!
- Introduce the possible relationships between controls and forms!
- Introduce the way how events are handled (EventHandler, EventArgs, key events)!
- How can new controls be created?
- Where can UserControls be used?

# Questions

- What is the connection between the Finalize method and the destructor? How are they working?
- Introduce the Dispose design pattern
- Show an example for the Dispose pattern (write code)!
- Write Windows Forms C# code that displays the pressed key in a messagebox.
- A Timer component (timer1) , a Label (label1) and two buttons (mStart and mStop) on a MenuStrip are placed on a Form. Write C# code that will start the timer if the mStart and stops it when the mStop button is pressed. When the timer is running a counter should be increased by one in every second and its value should be displayed in the label.

# References

- MSDN Library: „System.Windows.Forms Namespace”
  - > <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfssystemwindowsforms.asp>
- Implementing a Dispose Method
  - > <http://msdn2.microsoft.com/en-us/library/fs2xkftw.aspx>
- Quickstart tutorials (.NET 1 !!!)
  - > <http://samples.gotdotnet.com/quickstart/winforms/>