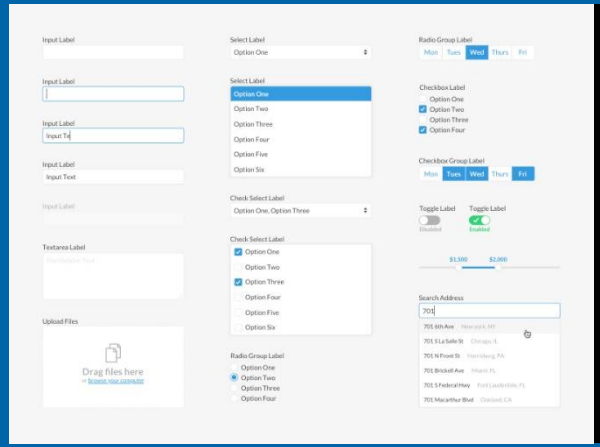
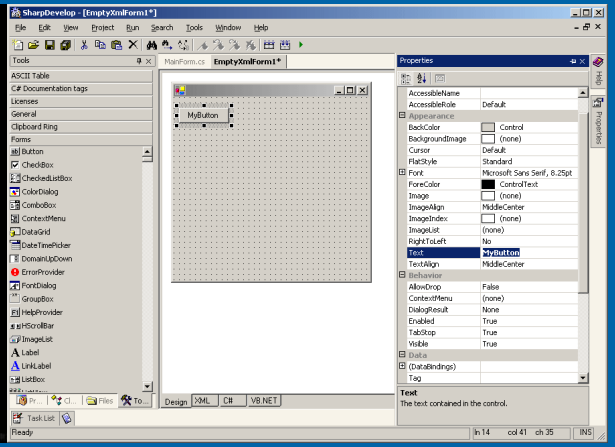
GUI-Programmierung: Designer und GUI-Elemente





BackgroundWorker und Fortschrittsanzeige

Verwendung des Designers und partieller Klassen

verschiedene GUI-Elemente

Evaluierung der Lehrveranstaltung

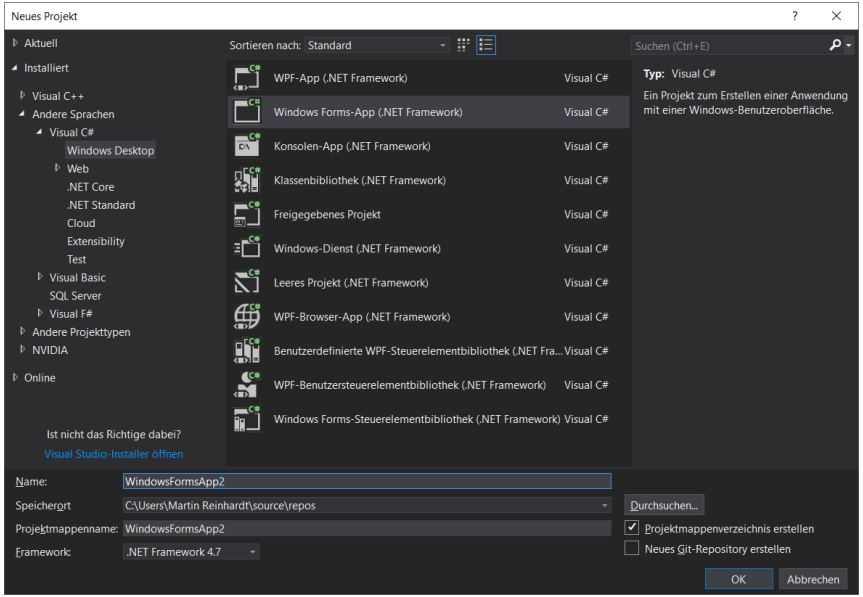
→ Bitte evaluieren Sie.

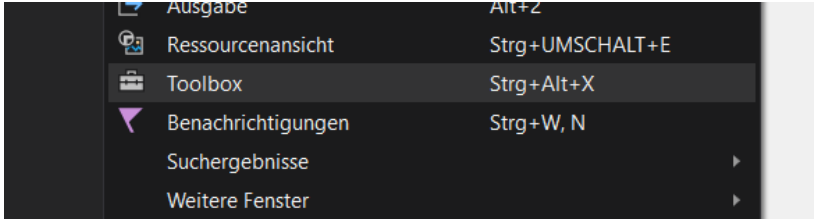
Einschub: Partielle Klassen

- Schlüsselwort: **partial**
- Aufteilen des Quellcodes einer Klasse
- Wird in GUI-Applikationen unter .NET massiv verwendet
- Aufteilen auf verschiedene Dateien

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

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

Windows Forms App mit Designer





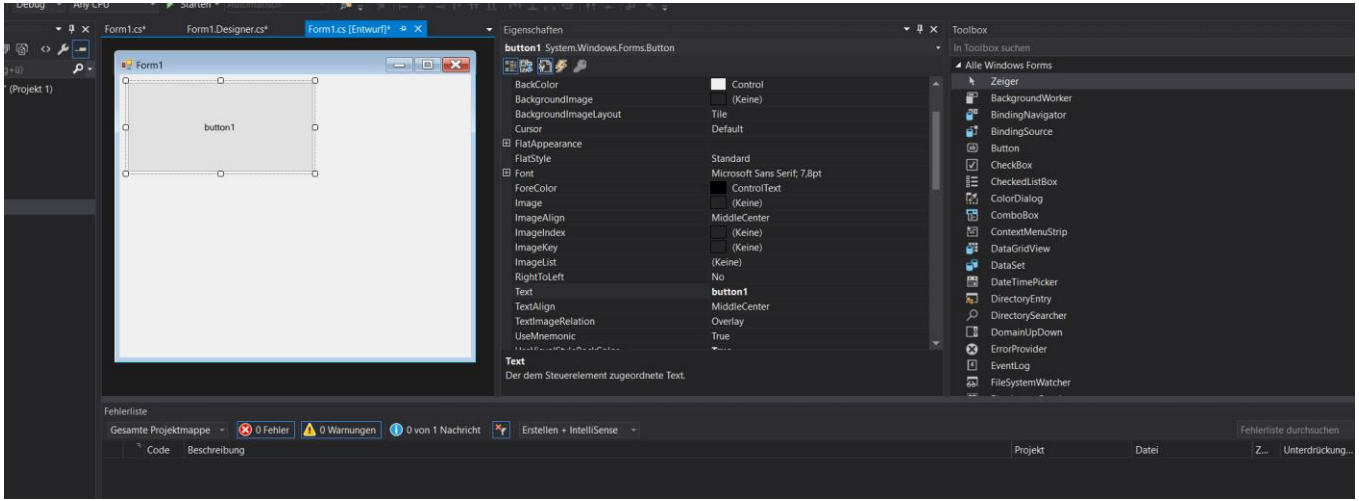
- Erstellen einer Windows-Forms-App
- Unter Ansicht → Toolbox
- Toolbox enthält die Designer-Elemente

Hauptprogramm und Einstiegspunkt

```
namespace WindowsFormsApp1
{
    static class Program
    {
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```

Standard-Programm für die Ausführung von Formular-Anwendungen

Designer-Nutzung



- ermöglicht schnelles Hinzufügen von Elementen in die Form
- Ändern aller relevanten Properties im Eigenschaften-Fenster
- genaue Positionierung der Elemente möglich

Erzeugter Code

Visual Studio erstellt zwei Dateien und drei Ansichten:

- Form1.cs
- Form1.Designer.cs
- Form1.cs [Entwurf]

empfohlener Workflow:

1. Hinzufügen eines Elementes im Designermodus
2. Einstellen aller Properties
3. Erstellen der zugehörigen Events und Methoden

Form1.Designer.cs

```
namespace WindowsFormsApp1
{
    partial class MyGuiForm
    {
        /// <summary>
        /// Erforderliche Designervariable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Verwendete Ressourcen bereinigen.
        /// </summary>
        /// <param name="disposing">True, wenn verwaltete Ressourcen gelöscht werden sollen; andernfalls False.</param>
        protected override void Dispose(bool disposing)
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        Vom Windows Form-Designer generierter Code

        private System.Windows.Forms.Button BigButton;
    }
}
```

Form1.cs

```
namespace WindowsFormsApp1
{
    public partial class MyGuiForm : Form
    {
        public MyGuiForm()
        {
            InitializeComponent();
        }
    }
}
```

```
private void InitializeComponent()
{
    this.BigButton = new System.Windows.Forms.Button();
    this.SuspendLayout();
    //
    // BigButton
    //
    this.BigButton.Location = new System.Drawing.Point(12, 12);
    this.BigButton.Name = "BigButton";
    this.BigButton.Size = new System.Drawing.Size(412, 346);
    this.BigButton.TabIndex = 0;
    this.BigButton.Text = "This is the Big Button";
    this.BigButton.UseVisualStyleBackColor = true;
    //
    // MyGuiForm
    //
    this.AutoScaleDimensions = new System.Drawing.SizeF(8F, 16F);
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.ClientSize = new System.Drawing.Size(800, 450);
    this.Controls.Add(this.BigButton);
    this.Name = "MyGuiForm";
    this.Text = "Form1";
    this.ResumeLayout(false);

}
```

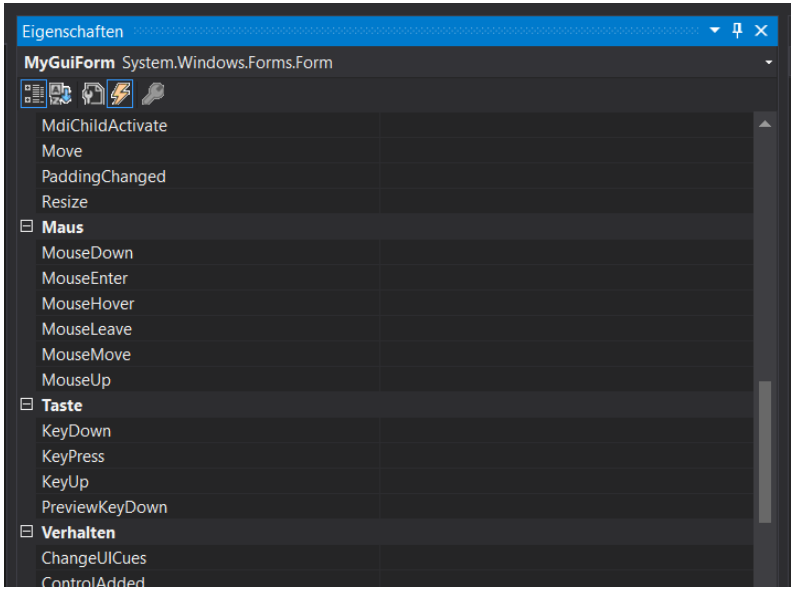
generierter Code

- enthält die Logik aller Steuerelemente
- **InitializeComponent()** ist die Methode des Designers

Hinzufügen von Event-Methoden

- Weg 1 : Doppelklick auf Steuerelement → üblichstes Event wird mit einem EventHandler versehen und eine Methode wird generiert

- Weg 2 : Nutzung der Eventliste im Eigenschaften-Fenster



Beispiel-Programm: >Big and Small Button Work<

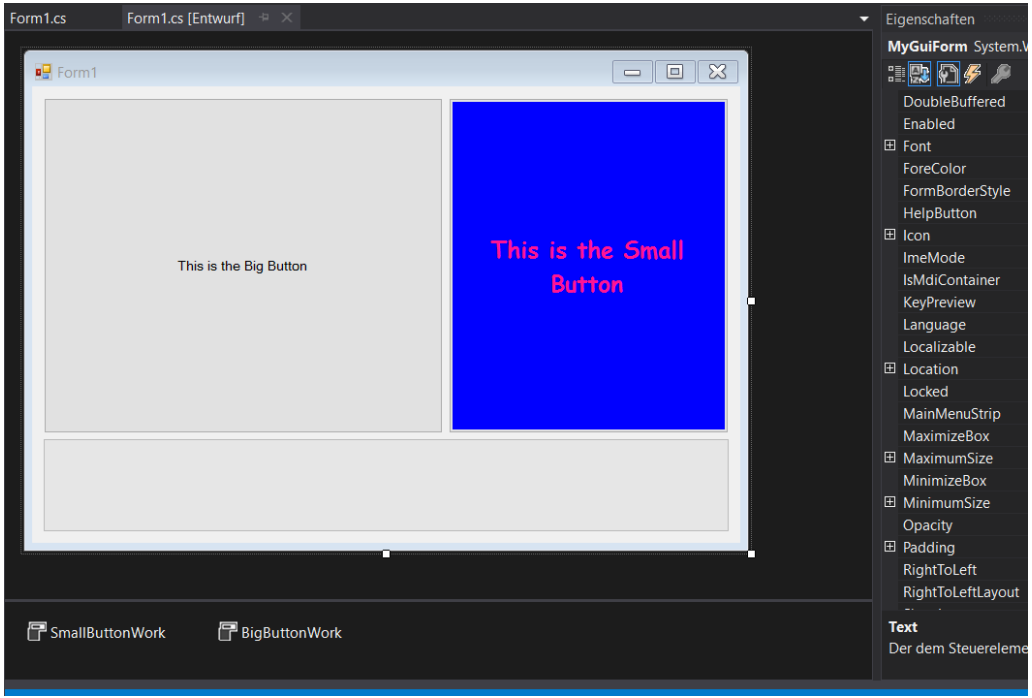
Elemente:

- kleiner Button
- großer Button
- Progress Bar

Logik:

- viel Arbeit beim Klicken des großen Buttons mit Feedback
- schnelles Ergebnis beim Betätigen des kleinen Buttons ohne Feedback
- jeweils Nutzung eines BackgroundWorkers
- Arbeit des großen Buttons soll beendet werden können

Kontrollelemente hinzufügen



- BackgroundWorker ebenfalls in der Toolbox
- sämtliche Bestandteile haben eigenen Namen
- Event-Methoden werden durch Designer erstellt

Events und Eventmethoden hinzufügen

```
public partial class MyGuiForm : Form
{
    public MyGuiForm()
    private void BigButton_Click(object sender, EventArgs e)
    private void MyGuiForm_Load(object sender, EventArgs e)
    private void SmallButton_Click(object sender, EventArgs e)
    private void SmallButtonWork_DoWork(object sender, DoWorkEventArgs e)
    private void BigButtonWork_DoWork(object sender, DoWorkEventArgs e)
    private void BigButtonWork_ProgressChanged(object sender, ProgressChangedEventArgs e)
    private void BigButtonWork_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
    private void SmallButtonWork_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
}
```

- alle Event-Handler werden automatisch erstellt
- Methoden werden automatisch mittels Event-Handler beim Event angemeldet

Kleiner Button

```
private void SmallButtonWork_DoWork(object sender, DoWorkEventArgs e)
{
    Thread.Sleep(1000);
}
```

```
private void SmallButtonWork_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    MessageBox.Show("Small Button Work is done");
}
```

```
private void SmallButton_Click(object sender, EventArgs e)
{
    SmallButtonWork.RunWorkerAsync();
}
```

- Arbeit wird durch eine Sekunde Pause simuliert
- BackgroundWorker verrichtet Arbeit und wird durch Button gestartet

Test

- großer Button ohne Funktion
- kleiner Button wird sauber ausgeführt
- Oberfläche bleibt bedienbar
- Mehrfaches Drücken des Buttons wirft Exception: **System.InvalidOperationException**

Eine Lösung findet sich in den Properties des BackgroundWorkers:

```
private void SmallButton_Click(object sender, EventArgs e)
{
    if(!SmallButtonWork.IsBusy)
    {
        SmallButtonWork.RunWorkerAsync();
    }
}
```

- verschiedene Lösungen würden hier passen
- weitere Möglichkeit:

```
SmallButton.Enabled = false;
SmallButtonWork.RunWorkerAsync();
```

```
MessageBox.Show("Small Button Work is done");
SmallButton.Enabled = true;
```

- Button wird deaktiviert, während der BackgroundWorker läuft:

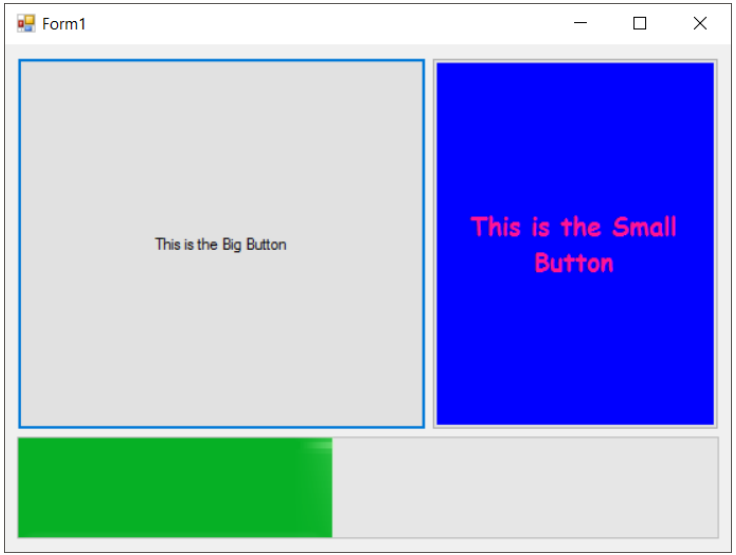
- großer Button soll Feedback mittels Progressbar liefern:

```
private void BigButtonWork_DoWork(object sender, DoWorkEventArgs e)
{
    for (int i = 0; i <= 100; ++i)
    {
        Thread.Sleep(100);
        BigButtonWork.ReportProgress(i);
    }
}
```

- zugehöriges Event:

```
private void BigButtonWork_ProgressChanged(object sender, ProgressChangedEventArgs e)
{
    myProgressBar.Value = e.ProgressPercentage;
}
```

- Ergebnis:



- Progressbar wieder auf 0 stellen:

```
private void BigButtonWork_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    MessageBox.Show("Big Button Work is done");
    myProgressBar.Value = 0;
}
```

- zweimaliger Klick auf großen Button soll den Worker abbrechen:

```
private void BigButton_Click(object sender, EventArgs e)
{
    if (BigButtonWork.IsBusy)
        BigButtonWork.CancelAsync();
    else
        BigButtonWork.RunWorkerAsync();
}
```

- Aufruf von CancelAsync setzt lediglich ein Flag: CancellationPending

- Flag muss in DoWork-Methode selbst ausgewertet werden

- angepasste DoWork-Methode:

```
private void BigButtonWork_DoWork(object sender, DoWorkEventArgs e)
{
    for (int i = 0; i <= 100; ++i)
    {
        Thread.Sleep(100);
        if (BigButtonWork.CancellationPending)
        {
            e.Cancel = true;
            break;
        }
        else
            BigButtonWork.ReportProgress(i);
    }
}
```

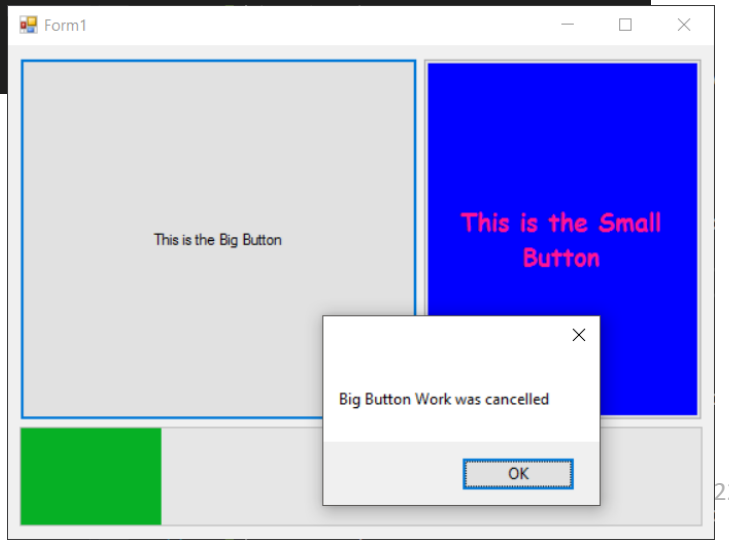
- Event-Flag muss noch gesetzt werden, damit **RunWorkerCompleted** auch den Grund für die Fertigstellung kennt

- RunWorkerCompleted Methode, angepasst:

```
private void BigButtonWork_RunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)
{
    if(e.Cancelled)
        MessageBox.Show("Big Button Work was cancelled");
    else
        MessageBox.Show("Big Button Work is done");
    myProgressBar.Value = 0;
}
```

- Resultat: funktioniert alles !

- Steuerelemente sehen je nach Plattform unterschiedlich aus



Eventreihenfolge

- der Empfänger eines Events ist immer vom jeweiligen Event abhängig:

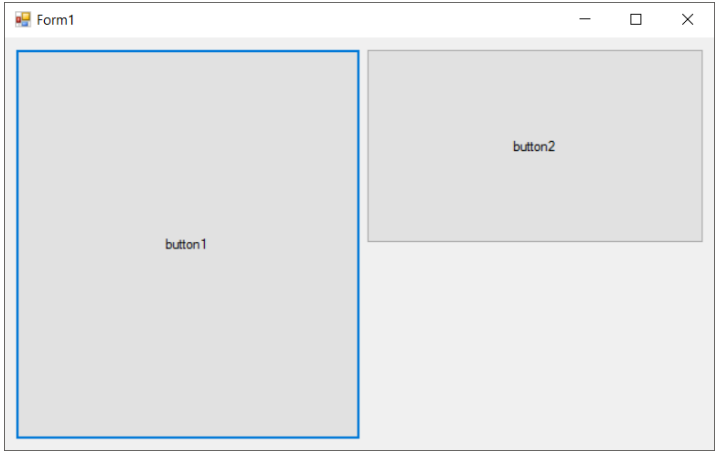
- MouseEvents: Event wird im Ziel ausgelöst

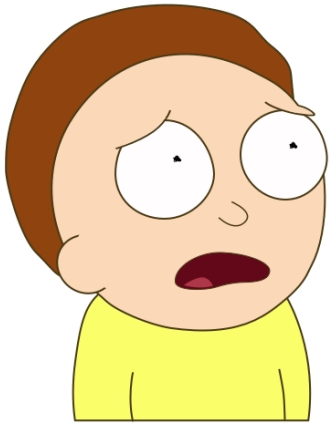
- KeyPress-Event: Immer das Element im Fokus ist

Wichtig ist auch die Event-Reihenfolge

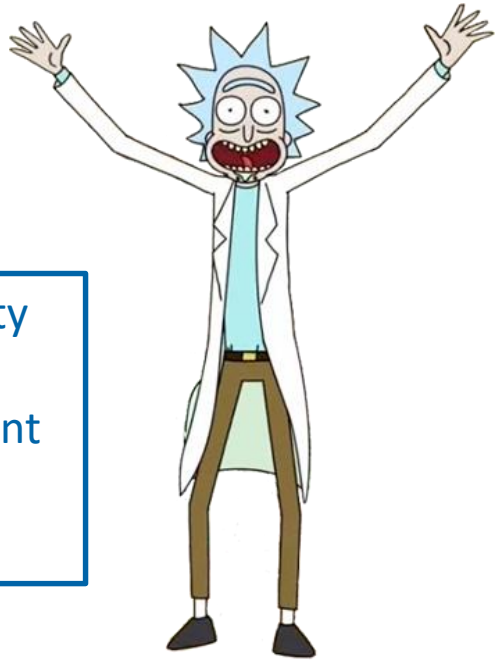
Event-Beispiel:

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
    MessageBox.Show("Form1 Key Press Event");
}

private void button1_KeyPress(object sender, KeyPressEventArgs e)
{
    MessageBox.Show("Button1 Key Press Event");
}

private void Form1_MouseDown(object sender, MouseEventArgs e)
{
    Text = "Form - " + e.Location.ToString();
}

private void button1_MouseDown(object sender, MouseEventArgs e)
{
    (sender as Button).Text = "Button - " + e.Location.ToString();
}

private void button2_KeyPress(object sender, KeyPressEventArgs e)
{
    MessageBox.Show("Button2 Key Press Event");
}
```



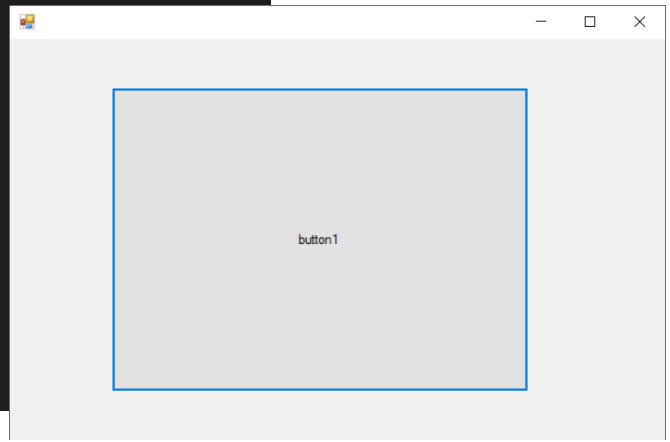


Was ist aber, wenn meine Form das KeyPress-Event fangen soll, damit ich Tastaturkürzel habe?



Das ist überhaupt kein Problem. Das Setzen der Property **KeyPreview=true** in der Form führt dazu, dass die Form das KeyPress-Event bekommt, bevor das fokusierte Control-Element dieses abarbeitet.

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
    switch(e.KeyChar)
    {
        case 'b':
            MessageBox.Show("Form: 'b'");
            break;
        case 'v':
            MessageBox.Show("Form: 'v'");
            break;
    }
}
private void button1_KeyPress(object sender, KeyPressEventArgs e)
{
    switch (e.KeyChar)
    {
        case 'n':
            MessageBox.Show("Button: 'n'");
            break;
        case 'v':
            MessageBox.Show("Button: 'v'");
            break;
    }
}
```



Verhindern der mehrfachen Ausführung

Nutzen der **Handled-Property** der **KeyPressEventArgs** zeigt, dass das Event bereits behandelt wurde.

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
    switch(e.KeyChar)
    {
        case 'b':
            MessageBox.Show("Form: 'b'");
            e.Handled = true;
            break;
        case 'v':
            MessageBox.Show("Form: 'v'");
            e.Handled = true;
            break;
    }
}
```

Standardverhalten bei Mausklicks:

Bei einem einzelnen Mausklick werden viele Events ausgelöst:

1. MouseDown
2. Click
3. MouseClick
4. MouseUp

Standardverhalten beim Tippen einer Taste

Beim Tippen einer Taste werden folgende Events ausgelöst:

1. KeyDown
2. KeyPress (nur von druckbaren Zeichen, Leerzeichen und Backspace)
3. KeyUp

Zusammenfassung

- partielle Klassen
- Nutzung des Designers
- Wdh: BackgroundWorker mit Abbruch und Fortschrittsanzeige
- Events von Tastatur und Maus
- Eventreihenfolge