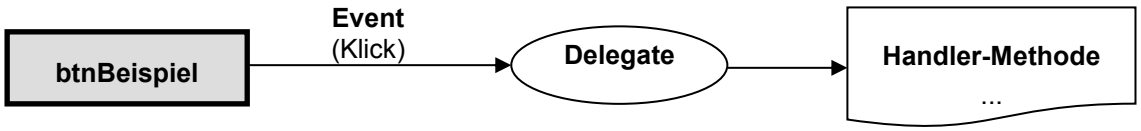
Windowsprogrammierung mit C# und .NET

1. Event – Handling unter C#

☞ **Erläutern Sie das Abfangen von Ereignissen unter C#!**
☞ **Welche Signatur besitzen Handler-Methoden im .NET Framework?**

Windows arbeitet **ereignisorientiert**. Unter C# sind für jedes grafische Steuerelement deshalb eine ganze Reihe von Events vordefiniert. Werden sie ausgelöst, so können diese abgefangen und in einer Handler - Methode abgearbeitet werden. Das Programm reagiert also auf Ereignisse.

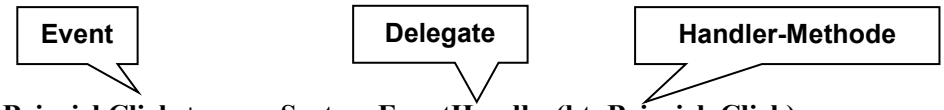
Beispiel: **Klick auf einen Button**



Klickt der Benutzer auf den Button btnBeispiel, so wird vom System der Click Event abgefeuert. Über ein vordefiniertes Delegate Objekt werden dieser Event abgefangen und die verknüpfte Handler Methode aufgerufen. Hier ist der Quellcode zu erstellen, der beim Klick auf den Button ausgeführt werden soll.

Zum Glück muss der Programmierer sich nicht um die Details kümmern. Events und Delegates sind für die Windows Steuerelemente und .NET Objekte bereits in der .NET Klassenbibliothek vordefiniert. Neben dem Click-Event sind für das Steuerelement Button dutzende weitere Events angelegt. Die Anwendung des Event Handling folgt jedoch Regeln:

Zum Verknüpfen des Events über das Delegate mit der Handler – Methode verwendet C# die folgende Syntax:




```
btnBeispiel.Click += new System.EventHandler(btnBeispiel_Click);
```

Beim Arbeiten mit dem Grafikeditor erledigt dies das Visual Studio allerdings automatisch.
Die Registrierung der Handler-Methode erfolgt dynamisch. Erst ab der Registrierung wird die Methode bei einem Klick aufgerufen.

Ebenso ist es möglich, eine Handler Registrierung zur Laufzeit zurückzunehmen. Die Syntax dazu sieht analog aus:

```
btnBeispiel.Click -= new System.EventHandler(btnBeispiel_Click);
```

Die Handler - Methode:

```
private void btnBeispiel_Click(object sender, System.EventArgs e)
```

Handler Methoden **geben immer void zurück** und **übernehmen immer zwei Parameter**. Den Sender des Events (hier btnBeispiel) als Typ object sowie ein EventArgs Objekt e (bzw. ein von EventArgs abgeleitetes Objekt). In diesem sind Zusatzinformationen zum Event gespeichert (bei Mouse Events z.B. die aktuelle Position der Maus oder ob gerade eine Maustaste gedrückt ist). Auch die Handler – Methode kann automatisch generiert werden.

2. Eine Beispielapplikation zur Dateiüberwachung

In einer Testapplikation zur Überwachung des Dateisystems sollen das Verwenden von Events sowie das Abfangen von Ausnahmefehlern geübt werden.

2.1 Die Ausgangssituation

In einem Firmennetzwerk gibt es einen Ordner WartungsFiles mit wichtigen Wartungs- und Fehlerprotokollen auf den mehrere User Zugriff haben. Treten neue Fehler oder Probleme auf, so werden vom bearbeitenden Techniker eine kurze Fehlermeldung sowie ein Protokoll über den Fehler und die Lösungsansätze hinterlegt. Damit alle Administratoren über die Änderungen im Ordner WartungsFiles Bescheid wissen, soll in Zukunft eine automatisierte Benachrichtigung erfolgen, sobald eine Datei im Ordner hinzugefügt, geändert oder gelöscht wird.
Vor der Implementierung der Funktionalität in einem Windows Dienst soll diese in einem kleinen Windows Forms Projekt als Prototyp getestet werden. Ihre Aufgabe besteht darin, dieses Testprojekt als Windows Forms Applikation in C# zu erstellen.

2.2 Das grafische Design der Testapplikation

- ☞ **Erstellen Sie ein neues Windows Forms-App Projekt namens Dateiüberwachung (.NET Framework 4.8x)!**
- ☞ **Ändern Sie den Namen des Forms im Projektmappenexplorer von Form1.cs auf FileWatcherForm.cs ab und bestätigen Sie die Codeänderungen im Meldungsfenster!**

Im Eigenschaftenfenster des Forms lassen sich die Properties des Forms grafisch editieren.

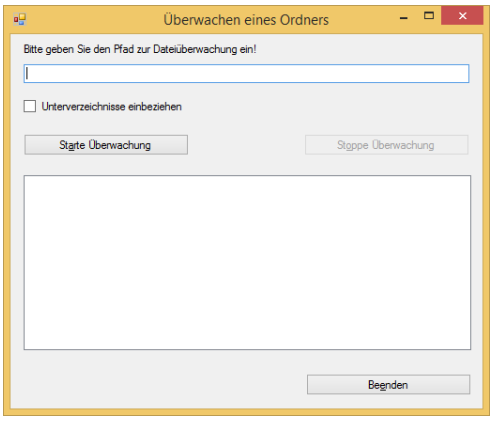
- ☞ **Ändern Sie die Größe auf 520 x 440 Pixel sowie die Überschrift auf „Überwachen eines Ordners“ ab und starten Sie den Form!**

Wie jedes grafische Steuerelement unter C# ist auch der Windows Form ein Objekt mit Attributen, Properties und Methoden. Seine Klassendefinition befindet sich im Quelltext.

- ☞ **Öffnen Sie den Form in der Quelltextansicht! Wo sind die von Ihnen geänderten Eigenschaften des Forms abgespeichert?**

Für den Prototypen ist folgende Funktionalität vereinbart:
Nach der Pfadeingabe des zu überwachenden Ordners startet man die Überwachung. Die Ergebnisse sollen in der Listbox angezeigt werden.

- ☞ **Fügen Sie die Steuerelemente entsprechend folgender Abbildung und Tabelle ein!**



Steuerelement	Name	Text
Label	lblPfad	Bitte geben Sie den Pfad zur Überwachung ein!
Textbox	txtPfad	--
Checkbox	chkSubFolder	Unterverzeichnisse einbeziehen
Button	btnStart	St&arte Überwachung
Button	btnStop	St&oppe Überwachung
Listbox	lstFiles	--
Button	btnBeenden	B&eenden

Das kaufmännische UND Zeichen „&“ beim Text der Buttons generiert einen Shortcut zur Steuerung der Buttons über die Tastatur.

- ☞ **Die Enabled Property des btnStop ist im Designer auf False zu setzen!**
- ☞ **Setzen Sie die Tabulatorreihenfolge über den Menüpunkt „Ansicht/ Aktivierreihenfolge“!**
- ☞ **Starten Sie die Applikation und testen Sie das Design!**

2.3 Die Funktionalität der Applikation

Für die Überwachung von Dateien und Ordnern stellt das .NET Framework die FileSystemWatcher Klasse bereit. Über Properties und Events der Komponente lässt sich das Datei und Ordnersystem auf einfache Art und Weise überwachen.
Folgende Properties und Events sind im FileSystemWatcher programmiert:

Property

bool EnableRaisingEvents	Schaltet die Überwachung ein oder aus.
string Path	Der Pfad des zu überwachenden Ordners.
bool IncludeSubdirectories	Legt fest, ob Unterverzeichnisse mit überwacht werden.

Event

Created, Changed, Renamed, Deleted	Je nach ausgeführter Aktion im überwachten Ordner wird der zugehörige Event abgefeuert.

Beim Abfangen des Events mit einer Handler Methode werden Zusatzinformationen zur Datei/ zum Ordner im FileSystemEventArgs Parameter namens e übergeben.

e.**FullPath** liefert den vollständigen Pfad und Namen, e.**Name** nur den Namen der geänderten Datei oder des geänderten Ordners ab.

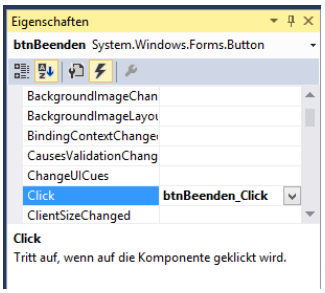
- ☞ **Fügen Sie dem Form in der Designansicht per Drag and Drop eine FileSystemWatcher Komponente hinzu (Komponentenansicht in der Toolbox)! Die Komponente erscheint unterhalb des Forms.**
- ☞ **Ändern Sie den Namen des Watchers in fsWatcher um und setzen Sie die EnableRisingEvents Property auf false! Der FileSystemWatcher ist nun einsatzbereit.**

2.3.1 Der Beenden Button

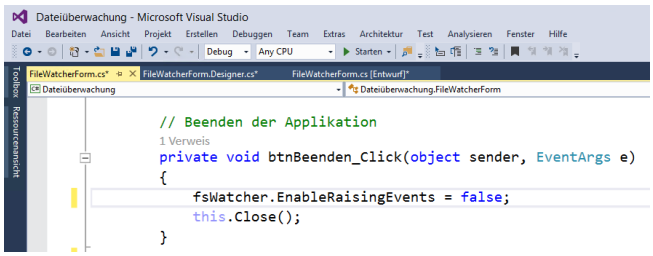
Beim Klick auf den Beenden Button soll die Applikation geschlossen werden. Das funktioniert am einfachsten über die Close( ) Methode des Forms.

Für einen Button sind diverse Events vorprogrammiert, welche der C# Programmierer in Handler - Methoden abfangen kann.

- ☞ **Öffnen Sie den Form in der Entwurfsansicht!**
- ☞ **Wählen Sie den Beenden Button durch Einfachklick aus!**
- ☞ **Schalten Sie im Eigenschaftenfenster auf Events um (Button mit dem Blitz)!**
- ☞ **Fangen Sie den Click Event des Buttons durch einen Doppelklick auf das entsprechende Event in einer Handler – Methode ab!**



Die Handler – Methode sowie die Verknüpfung über das Delegate werden vom Visual Studio Designer automatisch im Quelltext eingefügt.



In der Handler Methode ist das abfeuern der Events durch den fsWatcher auszuschalten.
Die Close( ) Methode schließt den Form. this liefert dabei eine Referenz auf das Formobjekt selbst, so dass der Form seine eigene Close( ) Methode aufrufen kann.

- ☞ **Erzeugen Sie eine Handler Methode für das FormClosing Event des Forms!**
- ☞ **Zeigen Sie ein Meldungsfenster beim Schließen des Forms an (MessageBox.Show(...))!**

2.3.2 Der Start Button

Mit dem Klick auf den Startbutton beginnt die Überwachung. Der User muss dazu erst einen Pfad für den zu überwachenden Ordner in der Textbox txtPfad eingeben.

☞ **Erstellen Sie die Handler Methode zum Klick Ereignis auf den Button btnStart!**

In der Methode setzt man die Path Property des fsWatchers auf den Text aus der Textbox.
Die Überwachung der angegebenen Datei über den FileSystemWatcher ist einzuschalten.
Ebenso deaktiviert man den Startbutton und aktiviert den Stoppbutton, um ein erneutes Starten der Überwachung zu verhindern.

```
//Startet die Überwachung des Ordners
private void btnStart_Click(…) …
{
  fsWatcher.Path = this.txtPfad.Text;
  …
  //Start und Stoppbutton aus/einschalten
  btnStart.Enabled = false;
  btnStop…
}
```

2.3.3 Der Stopp Button

☞ **Implementieren Sie die Handler Methode zum Klick Ereignis auf den Button btnStop!**

Beim Klick auf den Stopp Button stoppt man die Überwachung des Ordners über den fsWatcher und aktiviert bzw. deaktiviert wieder die Start- und Stopp Buttons.

```
//Stoppt die Überwachung des Ordners
private void btnStop_Click…
{
  //Überwachung des Ordners stoppen
  …
  //Start/Stoppbutton
aktivieren/deaktivieren

}
```

2.3.4 Die FileSystemWatcher (fsWatcher) Events

Genauso wie ein Button auf Klick Ereignisse reagieren kann, indem man das Ereignis in einer Handler Methode abfängt arbeitet auch der FileSystemWatcher mit Events und Handler Methoden. Beim FileSystem Watcher werden Events abgefeuert, wenn eine Datei oder ein Ordner erstellt, geändert, umbenannt oder gelöscht werden. Diese Events gilt es in entsprechenden Handler Methoden abzufangen und darauf zu reagieren.

☞ **Erstellen Sie die Handler Methode zum Changed Ereignis des FileSystemWatchers fsWatcher!**

In der Methode greift man auf die Listbox lstFiles zu und fügt deren Items Collection (eine Sammlung mit den Einträgen der Listbox) mit der Add(...) Methode einen neuen Eintrag hinzu.

```
// Abfangen der verschiedenen Änderungs-
// möglichkeiten im gewählten Verzeichnis
private void fsWatcher_Changed(…) …
{
  lstFiles.Items.Add("Geändert: " + e.FullPath);
}
```

Welcher Ordner oder welche Datei genau im überwachten Ordner geändert wurde kann über den Parameter e vom Typ FileSystemEventArgs herausgefunden werden. Die FullPath Property liefert dazu den vollständigen Pfad und Namen.

☞ **Implementieren Sie die Handler Methoden zu den Created, Deleted und Renamed Events des FileSystemWatchers fsWatcher! Gehen Sie dazu analog zum obigen Beispiel vor.**

☞ **Erstellen Sie den Ordner WartungsFiles unter K: und starten Sie die Überwachung!**
☞ **Testen Sie das Anlegen, Umbenennen, Ändern und Löschen von Dateien im überwachten Ordner!**

2.3.5 Die Checkbox zur Überwachung der Unterverzeichnisse

Die Checkbox dient zur Angabe, ob ein Benutzer die Unterverzeichnisse des zu überwachenden Ordners ebenfalls beobachten will oder nicht. Durch einen Klick auf die Checkbox wird das Häkchen gesetzt oder wieder entfernt. In der Checked Property speichert die Checkbox chkSubFolder dazu ihren aktuellen Status.

☞ **Fangen Sie das Klick Event auf die Checkbox in einer Handler Methoden ab!**

Die IncludeSubdirectories Property des FileSystemWatchers ist in der Methode auf den Wert der Checked Property der Checkbox zu setzen.

```
//Unterverzeichnisse mit überwachen
private void chkSubFolder_CheckedChanged(…)…
{
  //IncludeSubdirectories von fsWatcher setzen
  …
}
```

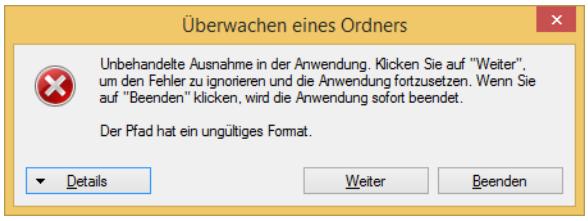
☞ **Testen Sie die Überwachung der Unterverzeichnisse des WartungsFiles Ordners!**

3. Exception Handling unter C#

Beim Gebrauch der Applikation sollte Ihnen ein Fehler auffallen. Testen Sie dazu folgende Vorgehensweise:

- Starten Sie die Applikation.
- Starten Sie die Überwachung, ohne ein Verzeichnis in der Textbox einzugeben.
- Setzen Sie das Häkchen „Unterverzeichnisse einbeziehen“.

Wie reagiert die Applikation?
Zur Laufzeit versucht der FileSystemWatcher den in der Path Property angegebenen Ordner zu überwachen. Die Unterverzeichnisse sollen mit einbezogen werden. Da in der Textbox kein Pfad angegeben ist scheitert er. Es wird eine Exception (Ausnahme) geworfen, die vom .NET Framework abgefangen wird. Zusätzlich wird eine Fehlermeldung angezeigt. Diese ist im Exception Objekt vom Typ System.ArgumentException enthalten.



Klickt man auf den Weiter Button, so wird das Programm „ohne Absturz“ beendet. Sinnvoller wäre es natürlich, dem User eine erneute Chance zu bieten seine Pfadangabe zu korrigieren. Dazu muss der Programmierer die Exception jedoch selbst abfangen.

3.1 Der allgemeine Ablauf beim Exception Handling

3.1.1 Die Exception Klassen

☞ **Beschreiben Sie wichtige Eigenschaften und Methoden der Exception Objekte!**

In der .NET Klassenbibliothek sind eine ganze Reihe von Exception Klassen enthalten, die alle von der Basisklasse System.Exception abgeleitet sind. Je weiter man in der Vererbungslinie nach unten geht, um so spezieller werden diese Exceptions. Sie besitzen folgende Properties und Methoden.

Properties

- HelpLink — Verknüpfung mit Hilfedatei,
- Message — Meldung, welche die aktuelle Ausnahme beschreibt,
- Source — Namen der Anwendung oder des Objekts, das den Fehler verursacht hat.

Methoden

- Konstruktor (Exception(...)) — Der Konstruktor ist überladen. Er kann als Parameter eine Fehlerbeschreibung übernehmen.
- ToString( ) — Liefert einen String mit der aktuellen Ausnahme zurück.

3.1.2 Das Fangen von Exceptions

☞ **Beschreiben Sie das Exception Handling unter C#!**
☞ **Erläutern Sie die Vorteile dieses so genannten „Structured Exception Handlings“!**

Unter C# gibt es vier Schlüsselwörter um Exceptions objektoientiert zu behandeln: try, catch, finally und throw.

Im Try – Block wird der Quellcode ausgeführt, der eine Ausnahme (Exception) werfen kann. Unter .NET ist eine ganze Klassenhierarchie von Exceptions enthalten, die alle von der Basisklasse System.Exception abgeleitet sind.
Eine Exception kann man auch selbst erzeugen und mit throw werfen. Dann sollte man dem Konstruktor aber eine Fehlermeldung übergeben!

Im Catch - Block fängt man die Ausnahme entsprechend ihres Datentyps (z.B. Exception, ArgumentException, ...) ab. Es können auch mehrere Catchblöcke mit unterschiedlichen Exceptiontypen angegeben werden. Dadurch kann man auf verschiedene Fehler unterschiedlich reagieren. Will man den Fehler nicht hier bearbeiten, so reicht man ihn mit throw an die aufrufende Methode weiter.

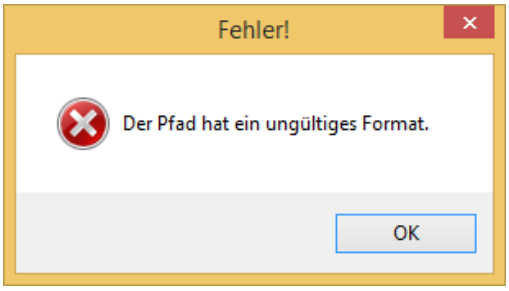
Der Finally - Block ist optional. Man muss ihn nicht verwenden. Hier programmiert man Code für „Aufräumarbeiten“, die immer ausgeführt werden müssen. Egal, ob der Try - Block fehlerfrei läuft oder eine Exception in catch gefangen wird, der Code im Finally - Block wird durchlaufen.

```
try
{
  //Auszuführender Code, der eine
  //Exception werfen kann
  ...
  ...
  //Eine Exception kann man aber auch
  //selbst erzeugen und werfen z.B.
  if (...)
     throw new Exception("Meldung…");
}
catch (Exception ex)
{
  //Hier wird die Exception gefangen,
  //Auszuführender Code um den Fehler zu
  //korrigieren
  ...
  ...
  //ggf. kann man die Exception an die
  //aufrufende Methode weitergeben, z.B.
  throw ex;
}
finally
{
  //dieser Code wird immer ausgeführt
  ...
}
```

Durch die Strukturierung und Trennung von Programmlogik (im Try - Block) und Fehlerbehandlungslogik (in den Catch - Blöcken) wird der Code wesentlich wartungsfreundlicher und übersichtlicher. Über das abgefangene Exception Objekt im Catch - Block erhält der Programmierer zudem noch Zusatzinformationen über die Art des Fehlers, der aufgetreten ist (z.B. ex.Message).

3.2 Der Einbau bei CheckedChanged

Zurück zum Fehler bei der Dateiüberwachung.
Setzt der User das Häkchen „Unterverzeichnisse Einbeziehen“, wenn kein Pfad angegeben sowie die Überwachung bereits gestartet sind, so wird eine Ausnahme (Exception) geworfen. Diese gilt es in der Handler Methode chkSubFolder_CheckedChanged(...) abzufangen.



☞ **Bauen Sie einen Try - Catch Block in der Methode chkSubFolder_CheckedChanged(...) ein!**
☞ **Eine Message Box soll dem User eine entsprechende Fehlermeldung liefern!**

Der bisherige Quelltext wird dazu in den Try Block verlagert.
Catch fängt das Exception Objekt ex. Zur Ausgabe der Fehlermeldung zeigt man den Inhalt der Message Property des Exception Objekts in einer Message Box auf dem Bildschirm an.
Die SelectAll( ) und Focus( ) Methoden des Textfeldes txtPfad setzen den Curser zur Eingabe eines neuen Pfades direkt in die Textbox. Der User kann seinen Eingabefehler korrigieren.

```
//Unterverzeichnisse mit überwachen
private void chkSubFolder_CheckedChanged(…)…
{
  try
  {
    //Unterverzeichnisse einbeziehen
     ...
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message, …, …, …);
    txtPfad.SelectAll();
    txtPfad.Focus();
  }
}
```

☞ **Testen Sie das Abfangen der Exception!**

3.3 Der Einbau bei btnStart_Click

Ein weiterer Eingabefehler ist im System leider immer noch möglich. Folgende Vorgehensweise verdeutlicht diesen Bug:

- Tippen Sie einen ungültigen Verzeichnisnamen in die Textbox!
- Starten Sie die Überwachung!
- Wie reagiert das System?

Beim Start kann der File System Watcher das angegebene Verzeichnis nicht finden. Er wirft eine Ausnahme, welche vom Programmierer abgefangen werden sollte.

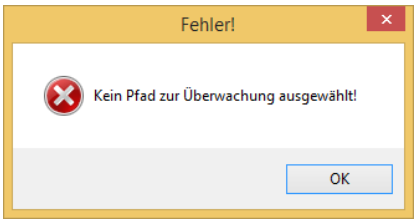
☞ **Ergänzen Sie die btnStart_Click(...) Handler Methode durch einen Try – Catch Block! Gehen Sie dazu analog zum vorherigen Beispiel vor!**

Der auszuführende Quelltext wird in einen Try Block gesetzt.

Im Catch Block gibt man die Meldung mit einer Message Box aus und setzt den Cursor zurück auf die Textbox txtPfad.

☞ **Testen Sie die Funktion des Exception Handlings!**

```
//Startet die Überwachung des Ordners
private void btnStart_Click(…)…
{
  try
  {
    //Quelltext ausführen
    ...
  }
  catch ...
  {
    //Fehler abfangen
    ...
  }
}
```



Ein letztes Fehlverhalten ist noch auszugleichen. Der File System Watcher lässt sich zwar nicht ohne Pfadangabe starten. Er führt dann keine Überwachung durch, wirft aber die falsche Fehlermeldung aus. Der Programmierer sollte also eine eigene Exception werfen, die den User darüber benachrichtigt.

☞ **Erstellen Sie ein eigenes Exception Objekt unter btnStart_Click(...) wenn die Pfadangabe leer ist!**

Gleich zu Beginn des Try - Blocks prüft man, ob der User einen Pfad eingegeben hat. Ist dies nicht der Fall, so wirft man mit throw ein neues Exception Objekt. Dem Konstruktor übergibt man dazu eine passende Fehlermeldung (siehe obige Grafik).

```
//Startet die Überwachung des Ordners
private void btnStart_Click(…)…
{
  try
  {
   if( this.txtPfad.Text == "")
     throw new Exception("Kein Pfad zur ...");
...
}
```

☞ **Starten und testen Sie die Applikation ausgiebig!**