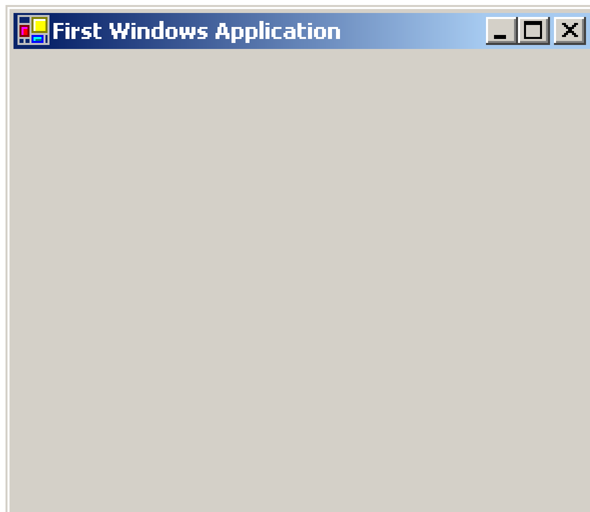
5 Windows Applikationen

5.1 Forms und Controls

Erzeugt man sich ein C# Projekt vom Typ Windows Applikation so wird folgender Quellcode generiert.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace WinAppFirst
{
   public class FormMain : System.Windows.Forms.Form
   {
      private System.ComponentModel.Container components = null;

      public FormMain()
      {
         InitializeComponent();
      }

      protected override void Dispose( bool disposing )
      {
         ...
      }


      #region Windows Form Designer generated code
      private void InitializeComponent()
      {
         this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
         this.ClientSize = new System.Drawing.Size(288, 229);
         this.Name = "FormMain";
         this.Text = "First Windows Application";
      }
      #endregion

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

Die Kommentare sind entfernt, der Name `Form1` wurde ersetzt durch `FormMain` und der Titel des Fensters wurde geändert. Das Programm kann sofort gestartet werden und zeigt ein Fenster mit bemerkenswerter Funktionalität.



Das Fenster kann verschoben werden, es kann minimiert und maximiert werden, man kann die Größe ändern und man kann es wieder schließen und das Programm damit beenden. Im .NET Framework ist die Funktionalität einer typischen Windowsapplikation in einer Klasse des Typs Form enthalten.

Windowsapplikationen konfrontieren uns mit einer Fülle von neuen Möglichkeiten der Sprache C# und des .NET Frameworks. Unter einem Framework versteht man eine Bibliothek von Klassen, welche jede Menge von der Funktionalität zur Verfügung stellen, die man zum Schreiben von Programmen benötigt. Die Klassen des Frameworks sind in unterschiedlichen Namensräumen gruppiert und verschachtelt, sodass sich dann voll ausgeschrieben Namen wie `System.Windows.Forms.Form` ergeben. Der Designer verwendet trotz der `using` Anweisung die Namen in der vollen Länge. Vieles davon werden wir zunächst einfach verwenden.

Die Kopfzeile der Klasse `FormMain`

```
public class FormMain : System.Windows.Forms.Form
```

unterscheidet sich von unseren bisherigen Klassen durch den Zusatz

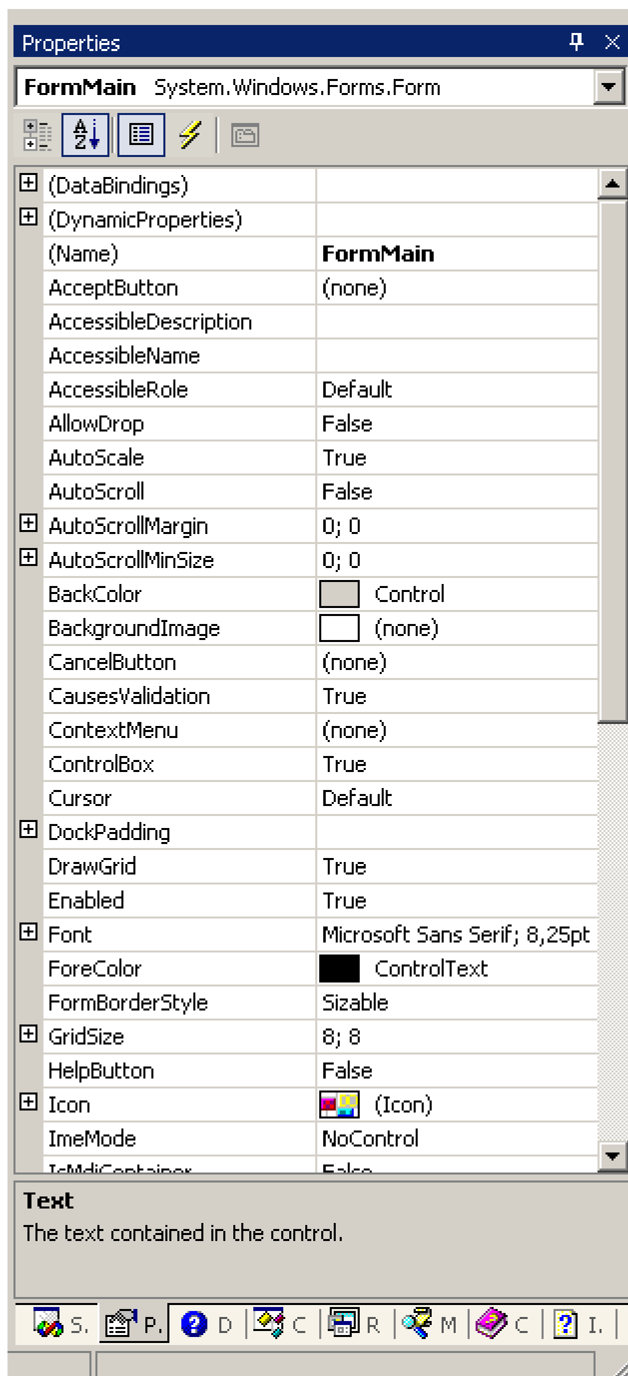
```
: System.Windows.Forms.Form
```

Das definiert die Klasse `FormMain` als Klasse, die von der Klasse `Form` abgeleitet ist. Das ist die Anwendung eines wichtigen Konzeptes in der OOP, der Vererbung. Eine Klasse kann von einer bestehenden Klasse Eigenschaften, Methoden etc. erben indem man sie von dieser bestehenden Klasse (man nennt das dann die Basisklasse) ableitet. Meist ist eine Frameworkklasse schon die n-te Generation einer Vererbungshierarchie.

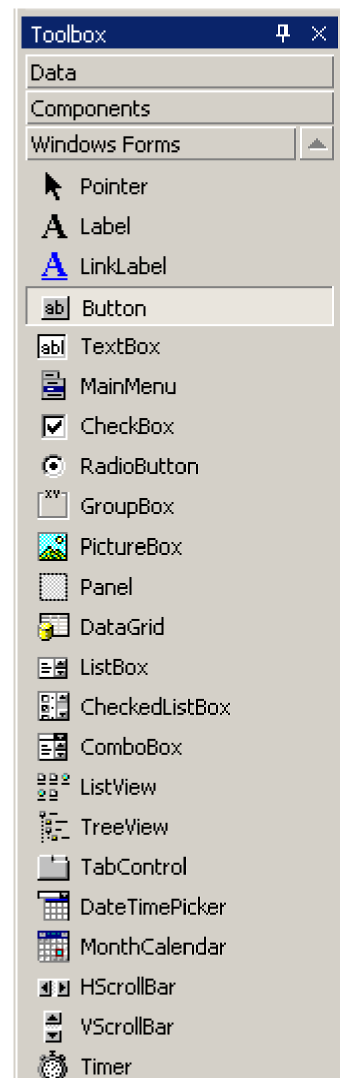
Man kann im Prinzip auch Windowsapplikationen zur Gänze mit dem Editor mühsam auskodieren. Die IDE (Integrierte Entwicklungsumgebung) öffnet aber eine Form standardmäßig in der "Designer-Ansicht". Ein Eigenschaftsfenster (Properties) zeigt uns eine Fülle von Eigenschaften einer Form, die man verändern kann und die viele Details des Fensters bestimmen. In unserem Beispiel wurden nur die Eigenschaften `Name` und `Text` geändert. `Name` ist der Name der Klasse. `Text` ist der Text im Titel des Fensters.

```
this.Name = "FormMain";
```

Das reservierte Wort `this` bezeichnet in einer Klasse die Klasse selber, also hier die Klasse `FormMain`. Meistens könnte man es weglassen, weil der Kontext eindeutig ist. Name ist eine Property der Klasse, `this.Name` legt diese Eigenschaft fest (set).



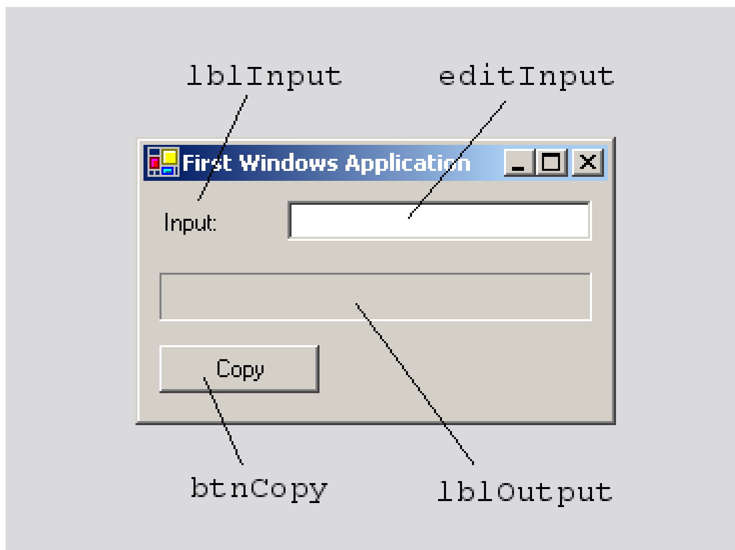
Property-Fenster



Toolbox

Windowsapplikationen verwenden eine Fülle von wohlbekannten so genannten Steuerelementen (*Controls*), die man in der Toolbox anwählen und auf der Form platzieren kann.

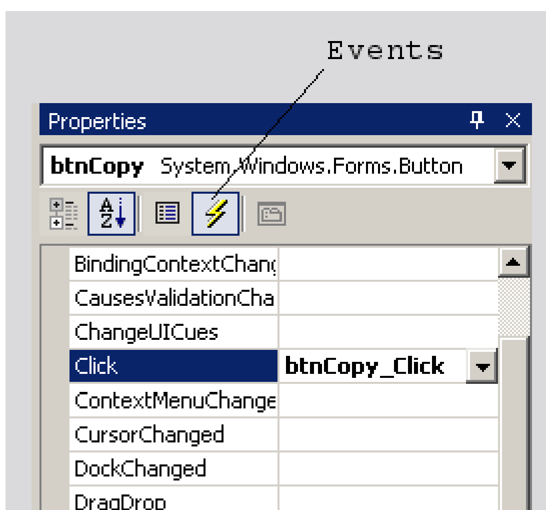
In diesem ersten Programm verwenden wir eine Textbox, zwei Labels und einen Button. Eine Textbox erlaubt die Eingabe von Text, ein Label dient zur Anzeige einer Beschriftung oder eines Informationstextes und mit einem Button kann man eine bestimmte Aktion auslösen.



Das Eigenschaftsfenster zeigt jeweils die Eigenschaften des ausgewählten (selektierten) Elements. Für alle Elemente wurde der Name geändert. Es ist wichtig ein einheitliches System für die Namen zu verwenden. Labels beginnen mit `lbl`, editierbare Textboxen mit `edit`, Buttons mit `btn`. Die zweite Änderung betrifft jeweils die Text Eigenschaft. Für das Label lblOutput ergibt sich die Vertiefung durch die Eigenschaft `BorderStyle = Fixed3D`.

5.2 Events

Windowsapplikationen werden von Ereignissen gesteuert. Im Wesentlichen wartet eine Windowsapplikation auf eine Interaktion des Benutzers. Das sind Mausklicks, Tastaturbefehle oder andere Eingaben. Diese Aktionen erzeugen Nachrichten (*messages*) die vom Betriebssystem in einer Nachrichten-Warteschlage (*message-queue*) gesammelt, den einzelnen laufenden Programmen zugeordnet und abgearbeitet werden. Im Programm kann man auf diese Nachrichten oder Ereignisse (*events*) reagieren. Neben Eigenschaften und Methoden sind Events eine weitere Funktionalität, welche in einer Klasse festgelegt werden kann und von Objekte zur Verfügung gestellt wird. Man kann die Events eines Objekts abonnieren. Dabei wird eine Methode angeben, die aufgerufen wird, wenn das Ereignis auftritt (*event handler*). Wenn sie im Designer ein *Control* doppelklicken, so wird damit der Code erzeugt, der das Voreingestellte Standard Ereignis abonniert und das Gerüst der Eventhandler-Methode erzeugt. Für einen Button ist der Maus-Klick das Standardereignis. Die bessere Variante ist es, das gewünschte Ereignis im Eventfenster auszuwählen.



Auch hier wird durch einen Doppelklicks in das zunächst leere Feld neben dem Namen des Ereignisses das Ereignis abonniert und eine Eventhandler-Methode eingefügt.

Um ein abonniertes Ereignis wieder entfernen, löschen Sie einfach den Text im Feld.

Programmcode zum abonnieren des Events:

```
this.btnCopy.Click += new System.EventHandler(this.btnCopy_Click);
```

Event-Handler:

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

}
```

Event-Handler haben in der Regel diese Bauart. Das erste Argument der Methode gibt Auskunft über den Sender des Events. Den Sender benötigt man dann, wenn nur ein Event-Handler für die Events mehrerer Controls verwendet wird. Das zweite Argument verweist auf ein Objekt des Typs `EventArgs` und enthält weitere Informationen zum Event. Das Event `MouseDown` liefert z.B. ein Objekt e, das Informationen zur Position des Mauszeigers und zur gedrückten Maustaste enthält.

Um uns vom Funktionieren des geschilderten Prinzips zu überzeugen, kopieren wir mit jedem Betätigen des Buttons "Copy" den Inhalt des Textfeldes in das Label `lblOutput`.

```
private void btnCopy_Click(object sender, System.EventArgs e)
{
   lblOutput.Text = editInput.Text;
}
```

5.3 Validierung

Ein Fenster mit den unterschiedlichen Controls ist die Schnittstelle zwischen Programm und Benutzer (*user interface UI*). Um ein Programm nicht mit falschen Daten zu versorgen ist es meist erforderlich, Eingaben auf ihre Gültigkeit zu überprüfen (*validate*).

5.3.1 Text in Zahl umwandeln

Eine solche Standardaufgabe ergibt sich z.B. sobald sie ein Eingabefeld für die Eingabe einer Zahl nutzen wollen. Genauso wie Console.ReadLine als Resultat Text liefert, liefert die Text-Eigenschaft den aktuellen Inhalt einer Textbox nur als Zeichenkette (`string`). Diesen Text muss überprüft werden, ob es sich um eine Zahl handelt und der text muss in eine Zahl umgewandelt werden. Die geeigneten Methoden dafür sind

```
public static double Double.Parse(string text);
public static int Int32.Parse(string text);
```

Statt `Double.Parse` und `Int32.Parse` kann man auch `double.Parse` und `int.Parse` schreiben. Entspricht der Text nicht den Regeln für eine gültige Darstellung der Zahl des gewünschten Zahlentyps, so kommt des zu einem Fehler, der eine so genannte Ausnahmebehandlung (*exception handling*) erfordert. Für den Typ double gibt es eine Variante `TryParse`, die ich bevorzuge, weil sie ohne Ausnahmebehandlung auskommt.

Ausnahmebehandlung benötigt man oft in Programmen. Es gibt dafür mit `try` .. `catch` eine eigene Kontrollstruktur.

Methode, um ganze Zahl validieren:

```
bool IsValidInteger(string text, out int newValue)
{
   newValue = 0;
   try
   {
      intValue = int.Parse(text);
      return true;
   }
   catch
   {
      return false;
   }
}
```

Offensichtlich kann man für die Argumente einer Methode eine Richtung angeben. Normalerweise wird für einfache Datenwerte in einer Methode mit einer Kopie des übergebenen Wertes gearbeitet, d.h. der Originalwert wird nicht verändert, die Richtung ist **in**[put]. Will man ein Argument in der Methode ändern, so muss man **out**[put] als Richtung angeben.

`try ... catch` funktioniert ähnlich wie `if ... else`:

Wenn es innerhalb des `try` - Blocks zu einem Fehler kommt, so wird der catch Block ausgeführt. Können die Aneisungen im `try` Block ohne Fehler abgearbeitet werden, so wird er `catch` Block nicht ausgeführt. Das Programm wird mit den Anweisungen nach dem `catch` Block fortgesetzt.

Genau denselben Code kann man analog auch für den Type double schreiben. Er entspricht nach außen ungefähr der Frameworkmethode `Double.TryParse`:

```
if (double.TryParse(editDouble.Text, NumberStyles.Float,
   NumberFormatInfo.CurrentInfo, out newDouble))
{
   doubleValue = newDouble;
}
```

`NumberStyles.Float` und `NumberFormatInfo.CurrentInfo` sind zusätzliche Angaben zur länderspezifischen gültigen Darstellung einer Zahl mit Dezimaltrennzeichen, Exponenten und Tausender-Trennzeichen.

5.3.2 Zeitpunkt der Validierung

Dafür gibt es zwei Ansätze, die sie in diversen Windows-Programmen auch erleben können.

1. Oft erfolgt die Übernahme von Daten beim Drücken eines Buttons ("Übernehmen", "*Apply*", "Ok"). Die Validierung kann jetzt so erfolgen, dass der Button die Überprüfung auslöst und im Fehlerfall ein Fenster mit einer entsprechenden Meldung (*message box*) anzeigt.
2. Die meist bessere Lösung ist, wenn unmittelbar nach der Eingabe die Überprüfung stattfindet. Diese Lösung erfordert eine ausführlichere Besprechung.

In einem Fenster kommt man von einem Control zum nächsten mit einem Mausklicks oder mit der Tabulator-Taste. Mit `Shift-Tab` geht's in die andere Richtung. Für die Bedienung mit der Tab-Taste hat der Programmierer dafür zu sorgen, dass die Controls in der richtigen Reichenfolge aktiviert werden, oder wie man auch sagt, den Fokus bekommen. Dies wird durch den `TabIndex` der Controls festgelegt. Über den `TabIndex` ergibt sich eine aufsteigende Nummerierung der Controls. Dieser Nummerierung folgt die Aktivierung der Controls über die Tab-Taste. Eine zusätzliche Eigenschaft `TabStop` legt fest, ob das Control dabei überhaupt berücksichtigt wird, oder nicht. Wenn Sie nichts tun, hat `TabStop` einen für das Control geeigneten Wert. Labels sind z.B. haben zwar einen `TabIndex`, aber keine `TabStop` Eigenschaft und werden übergangen. Sie können da zur Vertiefung ruhig etwas experimentieren.

Beim Übergang von einem Control zum anderen wird das `Validating` Event versendet. Dieses Ereignis nützt man zur Validierung. In der Ereignisprozedur für dieses Ereignis überprüft man die Gültigkeit der eingegeben Daten und nützt dann ein wertvolle Besonderheit des mitgelieferten Arguments `e` vom Typ `CancelEventArgs`. Setzt man

```
e.Cancel = true;
```

so wird der Vorgang abgebrochen und das gerade aktive Control bekommt den Fokus zurück.

```
private void editDouble_Validating(object sender,
              System.ComponentModel.CancelEventArgs e)
{
   double newDouble;
   if (double.TryParse(editDouble.Text, NumberStyles.Float,
                        NumberFormatInfo.CurrentInfo, out newDouble))
   {
      doubleValue = newDouble;
   }
   else
   {
      e.Cancel = true; // cancel validating -> go back to control
      MessageBox.Show("Enter a number!", "Error");
   }
}
```

5.4 Verwendung von verschiedenen Steuerelementen

5.4.1 Label

Das Label dient hauptsächlich zur Beschriftung von anderen Controls oder auch für die bloße Anzeige von Text. Bei gleicher Höhe und vertikalen Ausrichtung mit einem Textfeld passt die vertikale Ausrichtung der Schrift besser zusammen, wenn der Text im Label vertikal zentriert wird (`MiddleLeft`). Wählen Sie die Breite des Labels, groß genug, um auch den in andere Sprachen übersetzten Text anzeigen zu können, wenn die Software lokalisiert werden soll, d.h. in mehrer Sprachen zur Verfügung stehen soll. Verwendet man ein Label als Ausgabefeld, so kann man den `BorderStyle` auf `Fixed3D` umstellen, dann ist auch ein Label ohne Text sichtbar.

5.4.2 Textfeld (*textbox*) oder *Edit-Control*

Für die Eingabe einzelner Daten. Das Standardereignis ist das Event `TextChanged`, das bei jeder Änderung des Textes ausgelöst wird. Auf jedes neue oder gelöschte Zeichen währende der Eingabe will man in der Regel nicht reagieren, d.h. eine Ereignisprozedur für dieses Event wird man nicht definieren. Meist ist man mit dem `Validating` Ereignis bestens versorgt, um die Eingabe beim Verlassen des Steuerelements zu validieren und bei ungültigen Eingaben, den Fokus wieder an das Textfeld zurückzugeben.

5.4.3 Kontrollkästchen - *Checkbox*

Dieses UI Element ist gut geeignet als Schnittstelle zu einem Wahrheitswert, die Eigenschaft `Checked` zeigt an, ob das Kontrollkästchen ein Häkchen enthält oder nicht. Das Standardereignis ist `CheckedChanged`. Ein Mausklick wechselt den Zustand `von Checked` auf `UnChecked` und umgekehrt. Bevorzugt der Benutzer die Tastatur, so kann er mit der Leertaste den Zustand ändern.

Ist die Eigenschaft `ThreeState = true`, dann ist ein dritter undefinierter Zustand möglich, der über die Eigenschaft `CheckState` abgefragt werden kann.

5.4.4 Optionsfeld (*radio button*) und Gruppenfeld (*group box)*

Optionsfelder verwendet man, um aus mehreren Möglichkeiten eine auszuwählen. Optionsfelder treten daher immer in Gruppen auf und sind so implementiert, dass automatisch immer nur ein Optionsfeld ausgewählt ist, das dann durch einen Punkt markiert ist. Der Name *radio button* kommt auch tatsächlich von jenen Druckknöpfen auf alten Radios, mit denen man den Sender ausgewählt hat. Drückt man einen Knopf, so wechselt der zuvor gedrückte Knopf in die Ruheposition. Die Zusammenfassung zu einer Gruppe erfolgt durch die Form, auf der sie platziert sind, oder durch einen Container (`Panel`, `GroupBox` oder `TabControl`). Eine Anordnung untereinander ist besser als eine zeilenweise Anordnung, die Zahl der Alternativen sollte maximal sieben sein und schon beim Entwurf bekannt und langfristig stabil sein.

Das Standardereignis ist wie bei der CheckBox das Ereignis CheckedChanged. Dieses Ereignis wird zweimal ausgelöst, einmal von jenem Optionsfeld, das abgewählt wurde und einmal von jenem Optionsfeld, das nun das ausgewählte ist. Das Ereignis muss für alle Optionsfelder einer Gruppe angemeldet werden. Sinnvoll ist aber, für alle Felder die gleiche Ereignisprozedur zu verwenden um dort dann festzustellen, welche Option jetzt gewählt ist.

```
this.grpBoxView.Controls.Add(this.radioBtnSmallIcon);
this.grpBoxView.Controls.Add(this.radioBtnLargeIcon);
this.grpBoxView.Controls.Add(this.radioBtnDetails);
...
this.radioBtnSmallIcon.CheckedChanged += new
    System.EventHandler(this.radioBtnView_CheckedChanged);
...
this.radioBtnLargeIcon.CheckedChanged += new
    System.EventHandler(this.radioBtnView_CheckedChanged);
...
this.radioBtnDetails.CheckedChanged += new
    System.EventHandler(this.radioBtnView_CheckedChanged);

private void radioBtnView_CheckedChanged(object sender, EventArgs e)
{
    RadioButton radioBtn = (RadioButton) sender;
    if (radioBtn.Checked)
    {
       Debug.WriteLine(radioBtn.Name + " checked");
       // e.g. switch (radioBtn.Name) ...
    }
}
```

Mit der Anweisung

```
if (radioBtn.Checked)
{
}
```

wird nur auf jenes Ereignis reagiert, das von dem Optionsfeld ausgeht, das gerade ausgewählt wurde.

5.4.5 Listbox

In einer ListBox stellt man verschiedene Items zur Verfügung, die man auswählen kann. Die Items einer ListBox werden in Collection verwaltet, wie man sie im .NET Framework immer wieder antrifft. Eigenschaften deren Namen ein Plural ist, lassen praktisch immer auf eine Liste von Objekten schließen. Allgemein wird eine Sammlung von Objekten als *Collection* bezeichnet. Wichtige Methoden jeder Collection sind Add und RemoveAt, die Eigenschaft Count liefert die Anzahl der Elemente.

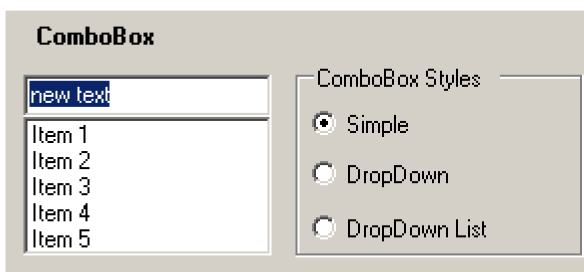
Sehr oft sind die ListBox Items einfach jene Liste von Wörtern, die in der ListBox sichtbar zur Auswahl angeboten werden. Man kann schon im Designer die Items mit einem Collection-Editor festlegen. Häufiger jedoch wird man die ListBox jedoch z.B. im Konstruktor oder im Load-Event-Handler der Form befüllen.

```
listBox.Items.Add("Item 1");
listBox.Items.Add("Item 2");
listBox.Items.Add("Item 3");
listBox.Items.Add("Item 4");
listBox.Items.Add("Item 5");
```

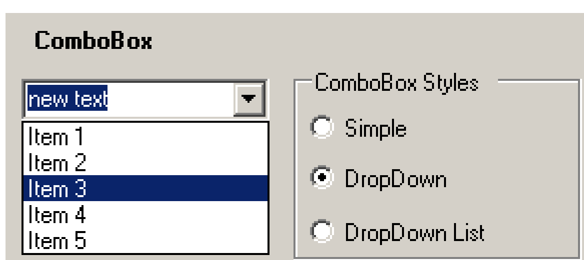
"Item x" steht hier natürlich nur als Verallgemeinerung für einen passenden Text. Das Standardereignis ist `SelectedItemChanged`. In der zugehörigen Ereignisprozedur kann man dann den Index der Auswahl oder das ausgewählte Item abfragen. Die Eigenschaft `SelectedIndex` liefert die fortlaufende Nummer des ausgewählten Items als int-Wert, wobei die Nummerierung, wie üblich bei 0 beginnt.

5.4.6 Kombinationsfeld (*ComboBox*)

Eine `ComboBox` ist zunächst einmal eine Kombination von `ListBox` und `TextBox`.



Mann kann eine Liste von Einträgen festlegen. Aus dieser Liste kann man eines auswählen, das dann in der Textbox angezeigt wird. Die Textbox ist aber editierbar, d.h. man kann Einträge verändern oder überhaupt beliebigen neuen Text eingeben. In .NET gibt es die ComboBox in drei Ausprägungen, welcher Stil verwendet wird, die durch die Eigenschaft `ComboBoxStyle` festgelegt. Der Stil Simple stellt die Listbox und das Textfeld getrennt dar. Der Stil `DropDown` bietet die Listbox Einträge in einer Klappliste an, die ein editierbares Textfeld integriert hat. Der Stil `DropDownList` reduziert die Combobox auf eine Listbox, mit einer aufklappbaren Liste. Bei diesem Stil können nur Einträge aus der Liste verwendet werden.



Auch für die Arbeit mit der ComboBox ist `SelectedItemChanged` das wohl wichtigste Ereignis. Die Auswertung erfolgt wie bei der Listbox. Die `Text`-Eigenschaft liefert den aktuell ausgewählten oder eingegebenen Text.

5.4.7 ListView

Dieses Control wird sehr oft eingesetzt. Was eine ListView alles kann, können Sie am Beispiel des Explorers studieren. Der Explorer verwendet eine TreeView um im Verzeichnisbaum zu navigieren und eine ListView, um die Dateien in einem Ordner anzuzeigen. Die Anzeige kann auf unterschiedliche Art erfolgen: Name der Datei und kleine Bilder (Icons) oder große Bilder oder in einer detaillierten Ansicht, die an eine Tabelle erinnert. Der Dateiname ist editierbar.

Die darzustellenden Items werden in einer Collection verwaltet, die Eigenschaft Items liefert eine Referenz auf diese Collection. Jedes Item wiederum hat eine Collection von SubItems. Beim Explorer ist der Dateiname das Item und die Größe, das Datum der letzten Änderung, die Attribute etc. sind Subitems.

Die verschiedenen Views:

SmallIcon

Für die Verwendung dieser Ansicht muss eine ImageList mit mindestens einem Bitmap (normal 16x16) vorbereitet werden. Angezeigt wird pro Item der Text des Items und das zugeordnete Bild als Icon.

LargeIcon

Man benötigt eine (weitere) ImageList mit mindestens einem Bitmap (normal 48x48). Angezeigt wird pro Item der Text des Items und das zugeordnete Bild als Icon.

List

Diese View zeigt die Items untereinander gemeinsam mit je einem Bild aus der ImageList für die kleinen Bilder an.

Detail

Die Details-View zeigt je ein Item in einer Zeile mit mehreren Spalten. Diese Spalten müssen zuerst vorbereitet werden. Die Kopfzeile der Spalten ergibt sich aus den Daten der `ColumnHeaderCollection Columns`, die auch im Designer mit einem Collection-Editor leicht erstellt werden kann. Wichtige Eigenschaften eines `ColumnHeaders` sind Text, das ist die Beschriftung, die Breite `Width` und `HorizontalAlimnment`. Das Definiert die Ausrichtung für die ganze Spalte. Optional kann man auch ein Gitter anzeigen lassen (`GridLines = true`).

Der Inhalt der einzelnen Spalten ist der Text der SubItems. Das erste SubItem (Index 0) enthält den gleichen Text wie das Item.

Hinzufügen von einem Item mit zwei SubItems:

```
ListViewItem item;
string itemText = "Adam";
// constructor with strItem as argument
item = new ListViewItem(itemText);
// image index in the image list, defines the icon
item.ImageIndex = 0;
// add subitem -> SubItems[1]
item.SubItems.Add("Paradies");
// add subitem  -> SubItems[2]
item.SubItems.Add("Blumengasse 12");
// add item to list view
listView.Items.Add(item);
```

Meist will man einen oder mehrere Einträge in der List-View auswählen können. Dazu schreibt man eine Ereignisprozedur zum Ereignis `SelectedIndexChanged`. Das folgende Beispiel ist aus einem Programm, das eine Liste von Zahlenwerten in einer ListView darstellt und es ermöglicht, die selektierte Zahl zu löschen oder in einem separaten Textfeld zu ändern. Die beiden zugehörigen Buttons sind nur aktiviert, wenn ein Item ausgewählt ist.

```
private void listView_SelectedIndexChanged(object sender,
System.EventArgs e)
{
  if (listView.SelectedIndices.Count > 0)
  {
    index = listView.SelectedIndices[0];
    btnUpdate.Enabled = true;
    btnDelete.Enabled = true;
    editValue.Text = String.Format("{0:f3}", values.GetItem(index));
  }
  else
  {
    btnUpdate.Enabled = false;
    btnDelete.Enabled = false;
  }
}
```

Wichtige Eigenschaften dazu:

`MultiSelect` bestimmt, ob man auch mehrere Items gleichzeitig auswählen kann, ist `FullRowSelect true`, so wird in der Detail-View immer die ganze Zeile markiert, `HideSelection` = `false` sorgt dafür, dass die Markierung nicht verschwindet, wenn die `ListView` den Fokus verliert.