HANSER

Erratum zu

Visual C# 2019 – Grundlagen, Profiwissen und Rezepte

von Jürgen Kotz

Print-ISBN: 978-3-446-45802-4
E-Book-ISBN: 978-3-446-46099-7

Anbei finden Sie die korrigierte Version der Seiten 238, 276-277 und 282.
Die geänderten Passagen sind grün markiert.

Beispiel 4.28: Eine Methode berechnet den Mittelwert eines übergebenen *double*-Arrays.

C#

```
double Mittelwert(double[] zahlen)
{
    double sum = 0;
    int z = zahlen.Length;
    for (int i = 0; i < z; i++)
    {
        sum += zahlen[i];
    }
    return(sum / z);
}
```

Die Verwendung:

```
double[] zahlen = {1.5, 2, 3.8, 0.7};  // Array mit vier Werten
double mw = Mittelwert(zahlen);        // Aufruf der Methode
MessageBox.Show(mw.ToString());        // zeigt den Durchschnitt "2"
```

HINWEIS: Wesentlich einfacher, als den Mittelwert per Hand zu berechnen, ist es, vordefinierte Methoden von Arrays zu nutzen. So liefert die Methode *Average()* das gewünschte Ergebnis.

C#

Die einfache Verwendung:

```
double[] zahlen = {1.5, 2, 3.8, 0.7};  // Array mit vier Werten
double mw = zahlen.Average();          // Aufruf der Extension-Methode Average
MessageBox.Show(mw.ToString());        // zeigt den Durchschnitt "2"
```

Rückgabe von Arrays

Methoden sind natürlich auch zur Rückgabe von Arrays in der Lage.

Beispiel 4.29: Die Methode liefert ein zweidimensionales *string*-Array.

C#

```
string[,] Kundenliste()
{
    string[,] Kunden = {{"Meier", "Berlin"}, {"Schultze", "Leipzig"},
                        {"Krause", "Bonn"}, {"Schneider", "München"}};
    return(Kunden);
}
```

Bei der Verwendung der Methode achten Sie darauf, dass der untere Feldindex immer mit 0 beginnt:

```
string[,] liste = Kundenliste();
MessageBox.Show("Name: " + liste[2, 0] + ", Wohnort: " + liste[2, 1]);
```

Quellcode

Die fett gedruckten Elemente im Code weisen auf Eigenschaften bzw. Methoden der *String*-Klasse hin, wie wir sie beispielsweise zum Herauskopieren (*Substring*) bzw. Suchen (*IndexOf*) eines bestimmten Zeichens benötigen.

```
public partial class Form1 : Form
{   ...
```

Zu Beginn deklarieren wir den Zeichensatz als Stringkonstante, in der alle erlaubten Buchstaben, Zahlen, Leerzeichen etc. enthalten sein müssen. Die Reihenfolge ist von untergeordneter Bedeutung (siehe Bemerkung am Schluss). Außerdem deklarieren wir eine Variable, die die Länge des Zeichensatzes ermittelt:

```
   private const string S0 = "abcdefghijklmnopqrstuvwxyz" +
                             "äöüABCDEFGHIJKLMNOPQRSTUVWXYZÄÖÜ1234567890 .,-?!";
   private int max = S0.Length;  // Anzahl der Zeichen
```

Beim Laden des Formulars werden der obere und untere Grenzwert der *NumericUpDown*-Komponente auf die positive bzw. negative Länge des Zeichensatzes gesetzt:

```
   private void Form1_Load(object sender, EventArgs e)
   {
      numericUpDown1.Maximum = max;
      numericUpDown1.Minimum = -max;
   }
```

Das Verschlüsseln und das Entschlüsseln des Textes wird von einer einzigen Funktion erledigt. Als Übergabeparameter erhält sie einen String *s* sowie die gewünschte (positive oder negative) Verschiebung *n*. Rückgabewert ist der verschlüsselte bzw. der entschlüsselte String:

```
   private string Codieren(string s, int n)
   {
      string s1 = string.Empty;  // zurückzugebender String
```

In der folgenden Schleife wird pro Durchlauf ein Zeichen aus dem übergebenen String „herauskopiert“ und seine Position im Zeichensatz gesucht. Anschließend wird das verschobene Zeichen berechnet und der Ergebnisstring wird stückweise wieder „zusammengebaut“:

```
       for (int i = 0; i < s.Length; i++)
       {
            char z = s.Substring(i)[0];  // i-tes Zeichen herauskopieren
            int pos = S0.IndexOf(z);     // Position im Zeichensatz suchen
            if (pos ==-1)                // Zeichen nicht gefunden
            {
                MessageBox.Show(z + " ist ein unzulässiges Zeichen!", "Warnung");
                break;
            }
            int posN = pos + n;           // auf neue Position verschieben
            // bei Überlauf wieder von vorn beginnen
            if (posN >= max)
            {
                posN = posN - max;
```

```
            }
            if (posN < 0)
            {
                posN = posN + max;      // ... bzw. hinten weitermachen
            }
            // korrespondierendes Zeichen ermitteln
            z = S0.Substring(posN)[0];
            s1 = s1 + z;                          // Rückgabestring zusammensetzen
        }
        return(s1);
    }
```

Der Funktionsaufruf beim Verschlüsseln:

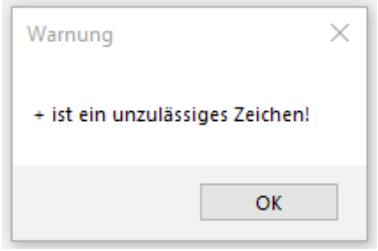
```
    private void button1_Click(object sender, EventArgs e)
    {
       textBox2.Text = Codieren(textBox1.Text, (int) numericUpDown1.Value);
    }
```

Der Aufruf beim Entschlüsseln:

```
    private void button2_Click(object sender, EventArgs e)
    {
       textBox3.Text = Codieren(textBox2.Text, (int) -numericUpDown1.Value);
    }
}
```

Test

Nach dem Programmstart haben Sie die Möglichkeit zu umfassenden Experimenten. Falls Sie ein nicht erlaubtes Zeichen eingeben, erfolgt ein Hinweis:

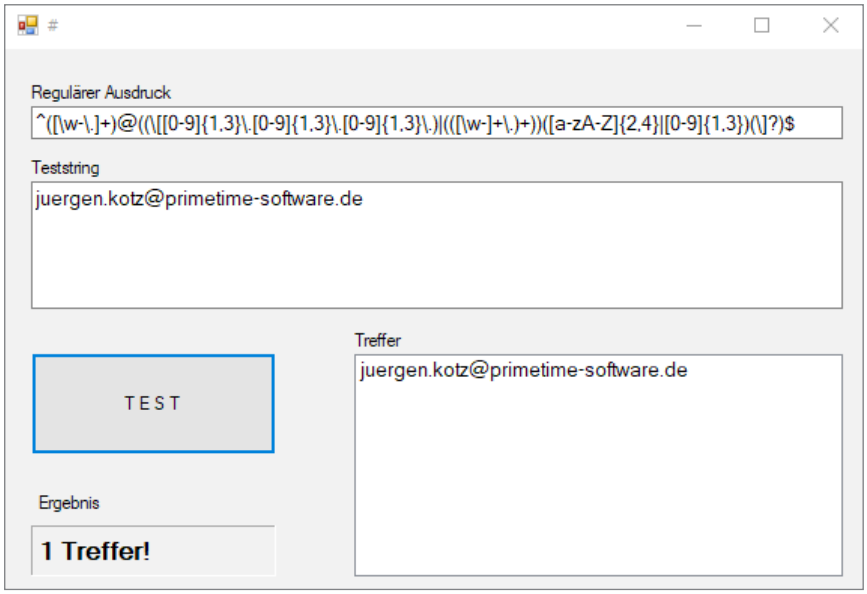


Bemerkungen

- Es liegt an Ihnen, den erlaubten Zeichenvorrat zu vergrößern bzw. einzuschränken. Dazu brauchen Sie lediglich die Stringkonstante *S0* zu ändern.
- Die „Knackfestigkeit“ des Verfahrens lässt sich deutlich steigern, wenn Sie die Zeichen innerhalb S0 nicht in alphabetischer Reihenfolge, sondern zufällig anordnen.
- Die Achillesferse unserer „Chiffriermaschine“ soll nicht verschwiegen werden: Der Hacker sucht im Text zunächst nach dem am häufigsten vorkommenden Zeichen und das ist mit hoher Wahrscheinlichkeit das verschlüsselte „e“. In obiger Laufzeitabbildung

Oberfläche

Die folgende Laufzeitansicht demonstriert die Gültigkeitsüberprüfung einer E-Mail-Adresse.



Quellcode

```
...
using System.Text.RegularExpressions;
 ...
 public partial class Form1 : Form
 {
     private string regex = null;
     private string text = null;

     private void button1_Click(object sender, EventArgs e)
     {
            regex = textBox1.Text;
            text = textBox2.Text;
            listBox1.Items.Clear();

            MatchCollection rxColl = Regex.Matches(text, regex);
            label1.Text = $"{rxColl.Count} Treffer!";
            foreach (Match m in rxColl)
            {
                listBox1.Items.Add(m.Value);
            }
     }
 }
```