Helma Spona

Windows PowerShell

Galileo Press

Auf einen Blick

Inhalt

Zeichenerklärung

Im Buch finden Sie viele Zusatzinformationen in grauen Kästen. Einige dieser Kästen sind mit Icons gekennzeichnet, die Ihnen anzeigen, welcher Art die Information ist:

[!] Warnhinweis: Die in diesen Kästen enthaltenen Informationen weisen auf potentielle Fehlerquellen hin.

[+] Tipp: In den Kästen, auf die dieses Zeichen aufmerksam macht, finden Sie kleine Tipps und Tricks, die Ihnen die Arbeit erleichtern.

[»] Hinweis: Diese Kästen enthalten hilfreiche Zusatzinformationen, z. B. Tipps, wie Sie die gezeigten Beispiele ergänzen können, oder Verweise auf Stellen mit weiterführenden Informationen zum Thema.

Skriptbeispiele zum Download

Alle Skriptbeispiele finden Sie auf der Website zum Buch **http://www.galileo-press.de/1385** unter der Rubrik »BuchUpdates«.

Wenn Sie nicht nur Befehle im Batch-Verfahren ausführen möchten, sondern Eingaben des Benutzers berücksichtigen möchten, müssen Sie mit dem Benutzer kommunizieren, indem Sie ihn Eingaben machen lassen und über Programmfortschritte informieren. Nicht nur im wahren Leben, sondern auch in der Programmierung gilt nämlich »Kommunikation ist alles«.

4 Kommunikation mit dem Anwender

Anders als im WSH können Sie aber nicht nur die integrierten Befehle zur Benutzerkommunikation nutzen, sondern auch die .NET-Objekte. Daher ist das .NET-Framework ein wesentlicher Aspekt, wenn es um die Gestaltung von Benutzeroberflächen geht.

4.1 Meldungen ausgeben und Werte einlesen

Meldungen haben Sie schon in den ersten Skripten immer wieder ausgegeben. Dazu gibt es das CmdLet `Write-Output`. Aber das, was Sie bisher dazu wissen, ist natürlich bei Weitem nicht alles. Nachfolgend soll daher auf die Details des CmdLets eingegangen werden. Darüber hinaus stellt die PowerShell aber auch zahlreiche weitere CmdLets zur Verfügung, mit denen Ein- und Ausgaben realisiert werden können.

4.1.1 Einfache Ausgaben mit Write-Output

Möchten Sie das CmdLet `Write-Output` für Ausgaben nutzen, sieht die einfachste Anweisung wie folgt aus:

```
Write-Output $text
```

In diesem Fall wird der Inhalt der Variablen `text` ausgegeben.

Sie können aber die erzeugte Ausgabe auch an ein anderes CmdLet übergeben und so bspw. die Ausgabe nicht an der Kommandozeile erfolgen lassen, sondern bspw. in eine Datei schreiben.

In diesem Fall übergeben Sie die Ausgabe einfach an das CmdLet `Out-File`. Mit dessen Parameter `-filepath` können Sie Name und Pfad der Zieldatei festlegen. Das folgende Skript gibt die Zeichenfolge `"test"` in die Textdatei **test.txt** aus.

```
#Skriptname: writeoutput.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Moeglichkeiten des CmdLets Write-Output
#Anmerkungen: keine

#Benoetigte Variablen
$text="Dies ist der auszugebende Text!"
$ausgabe=""
#Skriptbloecke und Funktionen

#Skriptinhalt
#einfache Ausgabe
Write-Output $text
#Ausgabe in eine Datei
Write-Output "test" |
   Out-File -filepath "G:\GAL_PowerShell\Bsp\K04\test.txt"
```

[!] Ist die Datei noch nicht vorhanden, wird sie erstellt. Wenn das angegebene Verzeichnis allerdings nicht vorhanden ist, kommt es zu einen Laufzeitfehler. Damit das Skript fehlerfrei läuft, müssen Sie also die Pfadangabe anpassen. Alternativ können Sie das Skript jedoch ergänzen und die Ausgabedatei im Skriptverzeichnis erzeugen. Dazu erstellen Sie sich eine Funktion, die das Verzeichnis des Skriptes zurückgibt. Diese Funktion rufen Sie dann auf, um ein gültiges Verzeichnis für die Ausgabe zu ermitteln.

[»] Wie Sie das Verzeichnis des aktuell ausgeführten Skriptes mit Hilfe des `System.Management.Automation.InvocationInfo`-Objekts ermitteln, wurde bereits in Abschnitt 3.5.3, *Rückgabewerte und Parameter für Skripte*, beschrieben. Das folgende Listing beinhaltet einfach nur eine Funktion `getPfad`, die den Pfad ermittelt und zurückgibt.

Die Funktion rufen Sie einfach auf und weisen ihr den Rückgabewert der Variablen `ausgabepfad` zu. Diese können Sie dann zusammen mit dem gewünschten Namen der Textdatei an den Parameter `-filepath` übergeben.

```
#Skriptname: writeoutput.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Moeglichkeiten des CmdLets Write-Output
```

```
#Anmerkungen: keine

#Benoetigte Variablen
$text="Dies ist der auszugebende Text!"
$ausgabepfad=""
#Skriptbloecke und Funktionen
function getPfad(
   [System.Management.Automation.InvocationInfo]$myInv=
   $myInvocation)
{
   $pfad=""
   $pfad=$myInv.get_MyCommand().Definition
   $pfad=Split-Path $pfad -parent
   return $pfad
}
#Skriptinhalt
#einfache Ausgabe
Write-Output $text
#Ausgabe in eine Datei
$ausgabepfad=getPfad
Write-Output "test" |
   Out-File -filepath ($ausgabepfad + "\test.txt")
```

[!] Ist die Datei schon vorhanden, wird sie überschrieben. Mehrere Ausgaben nacheinander in die gleiche Datei überschreiben daher immer alle vorhergehenden. Allerdings lässt sich auch das natürlich vermeiden, indem Sie den Parameter `-append` angeben. Der Parameter hat keinen Wert. Allein seine Existenz bewirkt, dass neue Ausgaben an das Ende der Datei angehängt werden.

```
Write-Output "test" |
   Out-File -filepath ($ausgabepfad + "\test.txt") -append
```

[+] Statt des CmdLets `Write-Output` können Sie auch den kürzeren Alias `echo` verwenden. Folgende Anweisung ist also ebenso korrekt:

```
echo "test" |
   Out-File -filepath ($ausgabepfad + "\test.txt")  -append
```

4.1.2 Ausgaben mit Write-Host

Mehr Möglichkeiten insbesondere für die formatierte Ausgabe bietet das CmdLet `Write-Host`. Damit können Sie nicht nur einfachen Text ausgeben, sondern auch Vorder- und Hintergrundfarbe bestimmen.

[!] Im Unterschied zu `Write-Output`, das die Ausgaben in den aktuellen Ausgabestrom schreibt, der nicht zwingend an der PowerShell-Kommandozeile ausgegeben werden muss, gibt `Write-Host` den Text immer an der Eingabeaufforderung aus. Wenn Sie in einer Funktion bspw. Ausgaben zur Kontrolle des Programmablaufs machen möchten, die nicht zum Rückgabewert der Funktion hinzugefügt werden sollen, verwenden Sie immer `Write-Host` anstelle von `Write-Output` oder `echo`.

Eine einfache Ausgabe erzeugen Sie aber auch hier, indem Sie den auszugebenden Text als unbenannten Parameter an das CmdLet übergeben:

```
Write-Host $text
```

[»] Die Ausgabe erfolgt nun standardmäßig wie die mit `Write-Output` erzeugten Ausgaben. Lediglich wenn Sie das Skript aus der PowerShellIDE starten, wird es mit weißer Schrift auf schwarzem Hintergrund ausgegeben, während die Ausgaben mit `Write-Output` in hellgrauer Schrift erfolgen.

Mit Hilfe der Parameter `foregroundColor` und `backgroundColor` können Sie die Schriftfarbe und Hintergrundfarbe definieren. Dazu geben Sie als Parameterwert eine gültige `system.consolecolor`-Konstante an.

Konstante	Farbe
`Black`	Schwarz
`DarkBlue`	Dunkelblau
`DarkGreen`	Dunkelgrün
`DarkCyan`	Dunkeltürkis
`DarkRed`	Dunkelrot
`DarkMagenta`	Violett
`DarkYellow`	Gold
`Gray`	Grau
`DarkGray`	Anthrazit
`Blue`	Blau
`Green`	Hellgrün
`Cyan`	Türkis
`Red`	Rot
`Magenta`	Pink
`Yellow`	Gelb
`White`	Weiß

Die Standardfarbe für die Schrift ist Weiß, für den Hintergrund Schwarz.

```
#Skriptname: writehost.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Moeglichkeiten des CmdLets write-host
#Anmerkungen: keine

#Benoetigte Variablen
$text="Dies ist der auszugebende Text!"
#Skriptinhalt
#einfache Ausgabe
Write-Host $text
#Ausgabe in blauer Schrift auf weissem Hintergrund
Write-Host $text -foregroundColor blue -backgroundColor white
Write-Host $text -foregroundColor green
```

Mit der vorstehenden Ergänzung wird die erste Ausgabe in weißer Schrift, die zweite mit blauer Schrift auf weißem Hintergrund und die dritte mit grüner Schrift auf schwarzem Hintergrund erzeugt.

Abbildung 4.1 Die erzeugten Ausgaben

[!] Wenn Sie das Skript direkt aus der PowerShellIDE ausführen, wird die Hintergrundfarbe nicht berücksichtigt.

Normalerweise werden alle Ausgaben, die Sie mit `Write-Output` oder `Write-Host` machen, untereinander in verschiedene Zeilen geschrieben. Das ist nicht immer günstig. Möchten Sie bspw. für längere Aktionen eine Art Fortschrittsanzeige ausgeben, wie sie unter DOS bspw. mit Punkten realisiert wurde, sollen die Ausgaben natürlich nebeneinander erscheinen.

Dazu können Sie den Parameter `-NoNewLine` verwenden. Geben Sie ihn an, erfolgt nach der Ausgabe kein Zeilenumbruch, sodass die nächste Ausgabe direkt dahinter in der gleichen Zeile erfolgt. Das folgende Skript zeigt dies und erzeugt auf diese Weise 1000 Punkte, die nacheinander ausgegeben werden.

[+] Wenn Sie sicherstellen möchten, dass die Ausgaben unter den Punkten wieder in einer neuen Zeile stehen, können Sie das CmdLet `Write-Host` auch ohne Parameter aufrufen. Dann wird nur ein Zeilenumbruch ausgegeben.

```
#Skriptname: writehost.ps1
#Autor: Helma Spona
```

```
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Moeglichkeiten des CmdLets write-host
#Anmerkungen: keine

#Benoetigte Variablen
...
$I=0
#Skriptinhalt
...
#Einfache Fortschrittsanzeige
for ($I=1;$I -lt 1000;$I++)
{
  Write-Host "." -NoNewLine -foregroundColor red
}
Write-Host
```

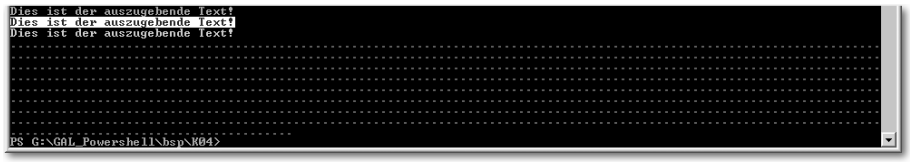

Abbildung 4.2 Die erzeugte Ausgabe der Schleife

4.1.3 Warnungen und Fehler ausgeben

Zwar können Sie mit Hilfe des CmdLets `Write-Host` auch formatierte Ausgaben machen und so Fehlermeldungen und Warnungen hervorheben. Aber das ist nicht unbedingt erforderlich. Sie können genauso gut spezielle CmdLets verwenden, die Warnungen und Fehlermeldungen ausgeben.

Für Warnungen gibt es das CmdLet `Write-Warning` und für Fehlermeldungen das CmdLet `Write-Error`. Der Warnung wird das Wort »WARNING:« vorangestellt, mit der Fehlermeldung werden auch der Skriptname und die Zeile ausgegeben, in der Sie das CmdLet aufgerufen haben.

```
Write-Warning "Dies ist eine Warnung!"
Write-Error "Das ist eine Fehlermeldung!"
```

[»] Falls Sie das Skript außerhalb der PowerShellIDE direkt in der PowerShell ausführen, werden Warnungen gelb formatiert. Innerhalb der PowerShellIDE werden Warnungen grau und Fehlermeldungen rot ausgegeben.

Abbildung 4.3 Die erzeugte Warnung und Fehlermeldung

4.1.4 Doppelte Ausgaben mit Tee-Object

Alle bisher gezeigten CmdLets geben die Ausgaben immer nur an ein Ziel aus: entweder an das nächste CmdLet in der Pipeline oder an der Kommandozeile. Möchten Sie eine Ausgabe aber bspw. sowohl an der Kommandozeile als auch in eine Datei ausgeben, wären dazu zwei Aufrufe erforderlich, wie dies das folgende Skript zeigt. Es gibt den auszugebenden Text in der Variablen `text` einmal in eine Datei aus und danach an der Kommandozeile als sichtbare Ausgabe.

```
#Skriptname: Teeobject.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die doppelte Ausgabe mit Tee-Object
#Anmerkungen: keine

#Benoetigte Variablen
$text="Dies ist der auszugebende Text!"
$ausgabepfad=""
#Skriptbloecke und Funktionen
function getPfad(
   [System.Management.Automation.InvocationInfo]$myInv=
   $myInvocation)
{
   $pfad=""
   $pfad=$myInv.get_MyCommand().Definition
   $pfad=Split-Path $pfad -parent
   return $pfad
}
#Skriptinhalt

#Ausgabe in eine Datei
$ausgabepfad=getPfad
Write-Output $text |
   Out-File -filepath ($ausgabepfad + "\test.txt")  -append
Write-Output $text
```

Mit Hilfe des CmdLets `Tee-Object` können Sie den gleichen Wert an zwei Stellen ausgeben. Geben Sie nur ein Ziel an, ist das zweite immer die Ausgabe an der

Kommandozeile. Die beiden vorstehenden `Write-Output`-Aufrufe könnten Sie also durch folgenden Aufruf ersetzen:

```
$text | Tee-Object  -FilePath ($ausgabepfad + "\test.txt")
```

Die Variable `text` wird über die Pipeline an das CmdLet übergeben. Über den Parameter `-FilePath` geben Sie den Namen und Pfad der Zieldatei für die Ausgabe an, wenn Sie die Ausgabe in eine Datei schreiben möchten.

Wenn Sie keine Pipeline verwenden möchten, können Sie das Eingabeobjekt, also den auszugebenden Inhalt, auch über den Parameter `-inputObject` angeben.

```
Tee-Object  -FilePath ($ausgabepfad + "\test.txt") -inputObject
$text
```

Alternativ können Sie die Ausgabe auch in eine Variable ausgeben. Dazu geben Sie den Namen der Variablen als Parameter `-variable` an. Über den Parameter `-variable` können Sie eine Variable erzeugen, in der die Ausgabe gespeichert wird. Dies zeigt die folgende Anweisung, die die Ausgabe einmal in die Variable `meineVariable` und zum anderen an der Kommandozeile ausgibt. Die Variable können Sie anschließend wie jede normale Variable weiterverwenden. Dass der Wert tatsächlich in der Variablen gelandet ist, zeigt die zweite Ausgabe mit der `echo`-Anweisung:

```
Tee-Object  -inputObject $text -Variable "meineVariable"
echo ("Ausgabe der Variablen: "  + $meineVariable)
```

[»] Anders, als die Hilfe zum CmdLet vermuten lässt, lässt sich die Augabe an der Kommandozeile aber nicht komplett unterdrücken, indem Sie zwei Ausgabeziele angeben. Sie können immer nur ein Ausgabeziel angeben, das zweite ist automatisch die Kommandozeile.

4.1.5 Benutzereingaben anfordern

In vielen bisher gezeigten Beispielen wurden Berechnungen mit konstanten Werten ausgeführt. Das ist in der Praxis in der Regel natürlich nicht sinnvoll. Mit Hilfe von Eingabeaufforderungen können Sie jedoch dem Benutzer ermöglichen, individuelle Werte zu bestimmen, mit denen die Berechnungen durchgeführt werden können. Auch Pfadangaben oder sonstige Parameter für Skriptaufrufe, die Sie benötigen, um Skripte zur Laufzeit konfigurieren zu können, lassen sich auf diese Weise vom Benutzer eingeben.

Das CmdLet, das Sie dazu benötigen, ist `Read-Host`. Es liest eine Benutzereingabe vom Prompt der Kommandozeile ein und ermöglicht auch die Ausgabe in eine Variable.

Der einfachste Aufruf könnte bspw. wie folgt lauten, indem Sie über den Parameter `-prompt` den Text für die Eingabeaufforderung bestimmen. Das CmdLet gibt dann den Wert zurück und gibt ihn aus, wenn Sie ihn nicht an eine Pipeline oder eine Variable weitergeben.

```
Read-Host -Prompt "Bitte geben Sie eine Zahl ein!"
```

Wenn Sie die Anweisung ausführen, wird die folgende Eingabeaufforderung ausgegeben:

```
PS G:\GAL_Powershell\bsp\K04> Read-Host -Prompt "Bitte geben Sie eine Zahl ein!"
Bitte geben Sie eine Zahl ein!: 2
2
```

Abbildung 4.4 Erzeugen der Eingabeaufforderung und Ausgeben der Eingabe

Wenn Sie das Skript bzw. die Anweisung in der PowerShellIDE ausführen, wird die Eingabeaufforderung als Eingabedialog angezeigt. Das liegt aber nicht am CmdLet, sondern eben an der Entwicklungsumgebung. Das CmdLet selbst erzeugt nur eine textbasierte Eingabeaufforderung an der Kommandozeile. **[«]**

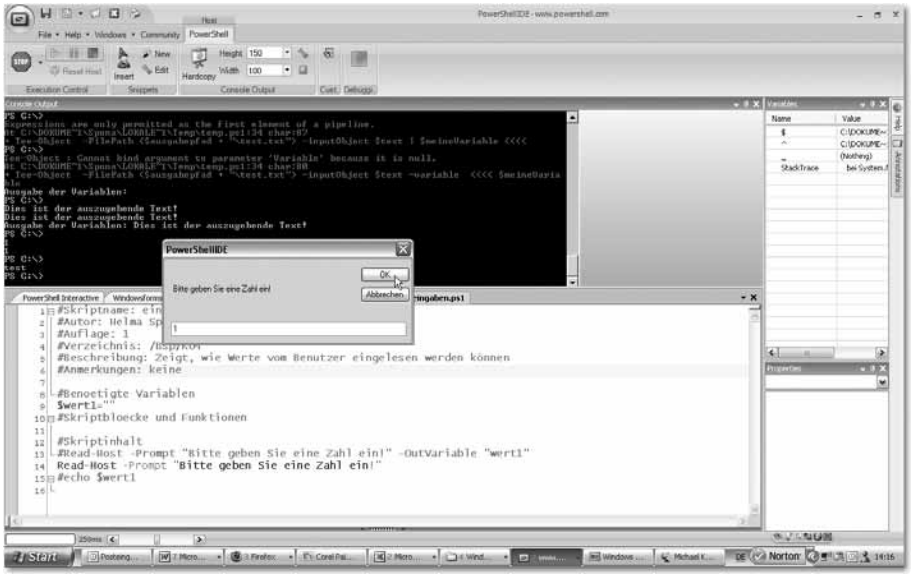

Abbildung 4.5 Die erzeugte Ausgabe der PowerShellIDE

Allerdings ist das natürlich nicht optimal. In Regel werden Sie Werte einlesen, um diese zu speichern und für weitere Berechnungen zu verwenden. Dazu können Sie optional den Parameter `-OutVariable` angeben. Als Wert legen Sie den Namen der Variablen fest, in dem die Eingabe gespeichert werden soll.

```
#Skriptname: eingaben.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt, wie Werte vom Benutzer eingelesen werden
koennen
#Anmerkungen: keine
#Benoetigte Variablen
$wert1=""
#Skriptbloecke und Funktionen

#Skriptinhalt
Read-Host -Prompt "Bitte geben Sie eine Zahl ein!"
   -OutVariable "wert1"
echo $wert1
```

[»] Die Variable müssen Sie vorher nicht zwingend definieren. Sie wird automatisch erzeugt. Allerdings macht es nichts, wenn die Variable schon vorhanden ist.

Trotz des Parameters wird der eingegebene Wert immer noch an der Kommandozeile ausgegeben, weil der Rückgabewert eines CmdLets immer ausgegeben wird, wenn Sie ihn nicht weiterverwenden. Möchten Sie den Aufruf nicht als ersten Teil einer Pipeline nutzen, können Sie daher auch den Parameter `-OutputVariable` weglassen und den Rückgabewert direkt einer Variablen zuweisen:

```
$wert1=Read-Host -Prompt "Bitte geben Sie eine Zahl ein!"
```

4.1.6 Kennworteingaben realisieren

Mit einem zusätzlichen Parameter können Sie auch Kennworteingaben erzeugen, bei denen die Eingabe verdeckt erfolgt. Dazu geben Sie zusätzlich den Parameter `-AsSecureString` an. Das CmdLet gibt dann eine Eingabeaufforderung aus und wandelt die eingegebenen Zeichen in Sternchen um. Die Eingabe wird auch in diesem Fall gespeichert, allerdings gibt das CmdLet dann ein `System.Security.SecureString`-Objekt zurück, das Sie so ohne Weiteres nicht ausgeben können.

```
...
#Benoetigte Variablen
$wert1=""
$kw=""
#Skriptbloecke und Funktionen

#Skriptinhalt
```

```
$wert1=Read-Host -Prompt "Bitte geben Sie eine Zahl ein!"
$kw=Read-Host -Prompt
   "Bitte geben Sie Ihr Kennwort ein!" -AsSecureString

echo $wert1
echo "Kennwort $kw"
```

Abbildung 4.6 Ausgaben des Skriptes – Die Kennworteingabe erfolgt verdeckt.

Wenn Sie das Skript direkt aus der PowerShellIDE ausführen, wird der Parameter `-AsSecureString` nicht berücksichtigt. Es erscheint stattdessen der normale Eingabedialog und die Ausgabe wird der Variablen `kw` zugewiesen und als normale Zeichenkette ausgegeben. **[«]**

Die Eingabe wird hier als `SecureString`-Objekt in verschlüsselter Form zurückgegeben, sodass Sie nicht direkt darauf zugreifen können. Wenn Sie bspw. prüfen möchten, ob das Kennwort einem vorgegebenen Kennwort entspricht, gibt es dazu im Prinzip nur eine praxistaugliche Möglichkeit. Sie konvertieren das Vergleichskennwort ebenfalls in eine verschlüsselte Zeichenkette und vergleichen die Eingabe damit. Um aus einer normalen Zeichenfolge eine verschlüsselte Zeichenkette zu erzeugen, gibt es das CmdLet `ConvertTo-SecureString`. Es akzeptiert ohne den Parameter `-AsPlainText` aber nur bereits kodierte Zeichenfolgen, keine normalen Zeichenfolgen. Sie müssen den Parameter dann aber zusätzlich angeben. Der Parameter `-Force` unterdrückt eine Warnung.

Mit der Funktion `verschluesselt` können Sie das vorgegebene Kennwort verschlüsseln und es dann mit dem eingegebenen Kennwort vergleichen.

```
#Skriptname: eingaben.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt, wie Werte vom Benutzer eingelesen werden
koennen
#Anmerkungen: keine

#Benoetigte Variablen
$wert1=""
$kw=""
$kwkorrekt=""
#Skriptbloecke und Funktionen
```

```
function verschluesselt([System.String] $strKW)
{
   $strSec=""
   $strSec = ConvertTo-SecureString $strKW
      -AsPlainText -Force
   return $strSec
}

#Skriptinhalt
$wert1=Read-Host -Prompt "Bitte geben Sie eine Zahl ein!"
echo $wert1
$kw=Read-Host -Prompt
   "Bitte geben Sie Ihr Kennwort ein!" -AsSecureString

$kwkorrekt=verschluesselt("test")
if ($kwkorrekt -eq $kw)
{
   echo "Kennwort korrekt!"
}
else
{
   echo "Kennwort falsch!"
}
```

4.2 Auf das .NET-Framework zugreifen

Bisher wurden immer nur alle Ein- und Ausgaben an der Kommandozeile gemacht. Das ist nicht sehr benutzerfreundlich und stellt auch eigentlich keinen wirklichen Fortschritt gegenüber dem WSH dar, der immerhin wenigstens grafische Dialogfelder mit Meldungen erzeugen konnte.

Was die PowerShell selbst mit CmdLets aber nicht ermöglicht, können Sie erreichen, indem Sie das .NET-Framework nutzen. Aber nicht nur dazu taugt es, sondern auch für andere Zwecke.

[»] An dieser Stelle lernen Sie die Basics für den Zugriff auf das .NET-Framework kennen. Im weiteren Verlauf der Kapitel werden dann weitere Einsatzmöglichkeiten an Beispielen gezeigt.

4.2.1 Einfache Meldungen ausgeben

Der wichtigste Namensraum des .NET-Frameworks in Zusammenhang mit der Benutzerkommunikation ist der Namensraum `System.Windows.Forms`. Er verfügt

über untergeordnete Namensräume und Klassen, die es Ihnen ermöglichen, Meldungen und Dialoge auszugeben und zu erzeugen.

Einige dieser Klassen sind statisch, wie die Klasse `System.Windows.Forms.MessageBox`. Deren Methode Show zeigt bspw. ein Dialogfeld mit einer Meldung an. Der einfachste Aufruf könnte daher wie folgt lauten:

```
[System.Windows.Forms.MessageBox]::Show("Hallo Welt!")
```

Diese Anweisung gibt den Text »Hallo Welt« als Meldung aus.

Abbildung 4.7 Die erzeugte Meldung

Sie können damit aber noch mehr machen, indem Sie den Titel, ein Symbol und die anzuzeigenden Buttons bestimmen oder den Rückgabewert verwerten. Zunächst gilt es aber, ein Problem zu lösen.

4.2.2 Notwendige .NET-Bibliotheken laden

Wenn Sie den Code in der PowerShellIDE ausführen, klappt das problemlos. Das Dialogfeld wird sichtbar. Anders ist das allerdings, wenn Sie den gleichen Code direkt in der PowerShell ausführen. Hier erhalten Sie dann eine Fehlermeldung:

```
Cannot find type [System.Windows.Forms.Messagebox]: make sure the
assembly containing this type is loaded.
```

Sie besagt einfach nur, dass die statische Klasse `System.Windows.Forms.Messagebox` nicht bekannt ist. Das wiederum liegt daran, dass diese Klasse nicht zu den .NET-Bibliotheken gehört, die die PowerShell automatisch zur Verfügung stellt. Sie müssen sie manuell laden.

Der Start des Skriptes aus der PowerShellIDE funktioniert nur deshalb, weil die PowerShellIDE intern standardmäßig einige Bibliotheken lädt, die dann schon geladen sind. **[«]**

Wenn Sie aber sicherstellen möchten, dass das Skript auch außerhalb der PowerShellIDE ausgeführt werden kann, sollten Sie die notwendigen Bibliotheken manuell laden. Das müssen Sie aber nur einmalig am Anfang des Skriptes tun und nicht jedes Mal, bevor Sie auf die Klassen des Namensraums zugreifen. Um

den Namensraum `System.Windows.Forms` zu laden, geben Sie vor dessen Verwendung folgende Anweisung ein:

```
[reflection.assembly]::LoadWithPartialName("System.Windows.Forms")
[System.Windows.Forms.MessageBox]::Show("Hallo Welt!")
```

Das Skript wartet dann, bis die Bibliothek geladen ist, und führt erst dann die folgenden Anweisungen aus. Nun wird die Meldung auch beim direkten Ausführen in der PowerShell angezeigt. Lediglich das Laden der Assembly wird bestätigt.

Abbildung 4.8 Das Laden der Assembly wird bestätigt.

[+] Falls Sie möchten, dass das Laden der Assembly nicht ausgegeben wird, können Sie das unterdrücken, indem Sie den Rückgabewert der statischen Methode `LoadWithPartialName` einfach einer beliebigen Variablen zuweisen:

```
$erg=[reflection.assembly]::LoadWithPartialName(
   "System.Windows.Forms")
```

4.2.3 Einen Titel angeben

Wenn Sie möchten, können Sie einen Titel angeben, der dann in der Titelleiste des Dialogs sichtbar wird. Den Titel übergeben Sie als zweiten Parameter an die `Show`-Methode. Sie können dort bspw. den Namen des Skriptes ausgeben, damit der Benutzer weiß, von welchem Programm die Meldung kommt.

Den Namen des Skriptes können Sie über das `System.Management.Automation.commandInfo`-Objekt ermitteln, das von der Eigenschaft `MyCommand` der Systemvariablen `myInvocation` zurückgegeben wird. Diese übergeben Sie dann einfach an die `Show`-Methode. Hier wird jedoch dem Namen des Skriptes noch die Angabe »PowerShell:« vorangestellt.

```
#Skriptname: msgbox.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Nutzung von Dialogfeldern
#   zur Anzeige von Meldungen

#Anmerkungen: keine
```

```
#Benoetigte Variablen
$erg=""
$meinName=$myInvocation.myCommand
#Skriptbloecke und Funktionen

#Skriptinhalt
$erg=[reflection.assembly]::LoadWithPartialName(
   "System.Windows.Forms")
[System.Windows.Forms.MessageBox]::Show("Hallo Welt!")
[System.Windows.Forms.MessageBox]::Show("Hallo Welt!",("PowerShell:"
+ $meinName) )
```

Abbildung 4.9 Ausgabe des Titels

Neben dem OK-Button, der standardmäßig angezeigt wird, wenn Sie nicht angeben, welche Schaltflächen sichtbar sein sollen, können Sie exakt bestimmen, welche Schaltflächen angezeigt werden sollen, indem Sie einen dritten Parameter an die `Show`-Methode übergeben. Als Wert müsse Sie eine `MessageBoxButtons`-Konstante übergeben. Allerdings müssen Sie deren numerische Entsprechungen verwenden, weil die PowerShell die Verwendung der Konstanten nicht unterstützt.

Angezeigte Buttons	Wert
Abbrechen (engl. **Abort**), **Wiederholen**, OK	2
OK	0
OK, **Abbrechen** (engl. **Cancel**)	1
Wiederholen, **Abbrechen** (engl. **Abort**)	5
Ja, **Nein**	4
Ja, **Nein**, **Abbrechen** (engl. **Cancel**)	3

Beachten Sie, dass es in englischen Windows-Versionen zwei verschiedene Abbrechen-Buttons gibt. Die einen heißen dort **Cancel**, die anderen **Abort**. Sie haben auch unterschiedliche Rückgabewerte und Aufschriften. Damit Sie die Konstanten und vor allem Rückgabewerte korrekt zuordnen können, wurden die englischen Aufschriften in Klammern angegeben. **[«]**

Möchten Sie also den **Ja**- und **Nein**-Button anzeigen lassen, müssen Sie somit den Wert `4` als dritten Parameter übergeben:

```
[System.Windows.Forms.MessageBox]::Show(
   "Hallo Welt!",("PowerShell:" + $meinName), 4)
```

Abbildung 4.10 Anzeige einer Meldung mit Ja- und Nein-Button

[»] Die Buttons haben mehr als nur eine optische Auswirkung. Sie können den Rückgabewert der `Show`-Methode ermitteln und so feststellen, auf welchen Button der Benutzer geklickt hat. Auf diese Weise können Sie bspw. vom Benutzer die Beendigung des Skriptes bestätigen lassen, wie dies das folgende Beispiel zeigt.

4.2.4 Rückgabewerte auswerten

Die `Show`-Methode gibt eine Zeichenkette aus, die angibt, auf welchen Button der Benutzer geklickt hat. Sie können also einfach den Rückgabewert mit einer möglichen Zeichenkette vergleichen, um festzustellen, mit welchem Button der Benutzer den Dialog geschlossen hat.

Der folgende Code zeigt eine Dialogbox an, die den Benutzer fragt, ob das Skript beendet werden soll. Klickt er auf **Ja**, wird das Skript mit `Exit` beendet.

Abbildung 4.11 Angezeigte Dialogbox mit Ja- und Nein-Button

```
...
if ([System.Windows.Forms.MessageBox]::Show(
   "Moechten Sie das Skript abbrechen?",
   ("PowerShell:" + $meinName), 4) -eq "Yes")
{
   echo "Abbruch durch den Benutzer ..."
   exit
}
else
{
```

```
   #hier folgt weiterer Code
   echo "Skript wird fortgesetzt ..."
}

...
```

Der Vergleich mit der Zeichenfolge (hier `"Yes"`) kann allerdings misslingen, wenn spätere Versionen der PowerShell vielleicht die Ausgaben gemäß der Betriebssystemsprache ausgeben. In deutschen Versionen würden Sie dann die Ausgabe `"JA"` erhalten, was natürlich dann dazu führt, dass der Vergleich nicht mehr gelingt. Sie sollten daher besser die numerischen Entsprechungen des Rückgabewertes verwenden. Welche Rückgabewerte zur Verfügung stehen, können Sie der folgenden Tabelle entnehmen:

Angeklickter Button	Rückgabwerte
Abbrechen (engl. **Abort**)	3
Abbrechen (engl. **Cancel**)	2
OK	1
Ja	6
Nein	7
Wiederholen	4
Ignorieren	5
kein Button	0

Möchten Sie prüfen, ob der Benutzer auf **Ja** geklickt hat, müsste damit der Vergleich wie folgt lauten:

```
if ([System.Windows.Forms.MessageBox]::Show(
   "Moechten Sie das Skript abbrechen?",
   ("PowerShell:" + $meinName), 4) -eq 6)
{ ...
```

4.2.5 Symbole anzeigen

Neben den Buttons können Sie auch Symbole bestimmen, die angezeigt werden sollen. Damit können Sie bspw. eine Fehlermeldung mit dem Fehlersymbol kennzeichnen und Fragen mit einem Fragezeichen versehen.

Die Konstanten für die Symbole geben Sie einfach als vierte Parameter für die `Show`-Methode an. Wenn Sie keinen Titel und oder keine Buttons angeben möchten, geben Sie für die Parameter einfach eine leere Zeichenfolge bzw. den Wert 0 an.

Symbol	Wert
K04_13 Frage	32
K04_14 Fehler/Stopp	16
K04_15 Warnung	48
K04_16 Info	64

Folgendes Beispiel stattet die schon vorhandene Frage mit einem Fragezeichen als Symbol aus und blendet danach Meldungen mit den anderen Symbolen ein:

```
...
if ([System.Windows.Forms.MessageBox]::Show(
   "Moechten Sie das Skript abbrechen?",
   ("PowerShell:" + $meinName), 4,32) -eq 6)
{
   echo "Abbruch durch den Benutzer ..."
   exit
}
else
{
   #hier folgt weiterer Code
   echo "Skript wird fortgesetzt ..."
}
[System.Windows.Forms.MessageBox]::Show("Fehler!","",0,16)
[System.Windows.Forms.MessageBox]::Show("Info!","",0,64)
[System.Windows.Forms.MessageBox]::Show("Warnung!","",0,48)
```

Abbildung 4.12 Die Frage mit dem definierten Symbol

[+] Wenn Sie Meldungen grundsätzlich in Dialogfelder ausgeben möchten, sollten Sie sich für jeden Meldungstyp eine eigene Funktion erstellen und diese in einer externen Skriptdatei ablegen. Wenn Sie diese Skriptdatei dann mit Ihren Skripten verknüpfen, können Sie mit wenig Aufwand die entsprechenden Meldungen erzeugen. Die einzubindende Skriptdatei könnte dazu bspw. wie folgt lauten:

```
#Skriptname: wichtigeFunktionen.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp
#Beschreibung: Wichtige Funktionen
#Anmerkungen: keine

#Benoetigte Variablen

#Skriptbloecke und funktionen
...
function Warnung([System.String]$strText)
{
   return [System.Windows.Forms.MessageBox]::Show(
      $strText,("PowerShell: Warnung"),0,48)
}

function Fehler([System.String]$strText)
{
   return [System.Windows.Forms.MessageBox]::Show(
      $strText,("PowerShell: Fehler"),0,16)
}

function Fehler([System.String]$strText)
{
   return [System.Windows.Forms.MessageBox]::Show(
      $strText,("PowerShell: Frage"),4,32)
}

#Skriptinhalt
$erg=[reflection.assembly]::LoadWithPartialName(
   "System.Windows.Forms")
```

Wenn Sie am Ende der Datei bereits die `System.Windows.Forms-Assembly` laden, brauchen Sie sich darum auch nicht mehr kümmern, wenn Sie diese Datei in alle Ihre Skripte einbinden.

In den Skripten, in denen Sie die Funktionen `Warnung`, `Fehler` und `Frage` aufrufen möchten, brauchen Sie dann nur noch die Datei mit den Funktionen einbinden. Details dazu finden Sie in Kapitel 3, *Sprachgrundlagen*.

Fügen Sie dazu einfach in das Skript vor dem Aufruf der Funktionen, am besten am Anfang des Skriptes, folgende Anweisungen ein:

```
#Laden der Hilfsfunktionen
$bibpfad= Split-Path (Split-Path ($myInvocation.get_
```

```
MyCommand().Definition) -parent)  -parent
. ($bibpfad + "\wichtigefunktionen.ps1")
```

[!] Sie müssen sicherstellen, dass sich die Datei `wichtigefunktionen.ps1` im übergeordneten Verzeichnis befindet. Möchten Sie das Skript aus der PowerShellIDE ausführen, muss es sich auch in dem Temp-Verzeichnis befinden, in das die PowerShellIDE die Skripte temporär kopiert. Standardmäßig ist dies das Verzeichnis **C:\Dokumente und Einstellungen***Benutzername***\Lokale Einstellungen\Temp** des aktuellen Benutzers, wenn Sie Windows auf Laufwerk C installiert haben.

4.2.6 Dateiauswahldialoge anzeigen

Die PowerShell selbst bietet leider ebenfalls keine Möglichkeit, eine Datei mit Hilfe eines Dialogs auszuwählen. Aber auch dazu können Sie das .NET-Framework verwenden. Die Klasse `Windows.Systems.Forms.OpenFileDialog` ist eine Ableitung der Basisklasse `Windows.Systems.Forms.FileDialog`. Sie stellt einen Datei-Öffnen-Dialog zur Verfügung, den Sie über verschiedene Eigenschaften konfigurieren können. Die `ShowDialog`-Methode gibt einen Wert zurück, aus dem Sie erkennen können, wie der Dialog geschlossen wurde. Über die Eigenschaft `FileName` können Sie nach Schließen des Dialogs den ausgewählten Dateinamen ermitteln.

[»] Basisklassen sind Klassen des .NET-Frameworks, die Sie nicht direkt instanziieren können. Sie können sie verwenden, um daraus eigene Klassen zu erzeugen. Von bestimmten Basisklassen stellt das .NET-Framework bereits solche abgeleiteten Klassen zur Verfügung. Ein Beispiel ist eben die Klasse `OpenFileDialog`.

Anders als bei der `MessageBox`-Klasse müssen Sie die Klasse `OpenFileDialog` aber instanziieren, das heißt, Sie müssen ein Objekt daraus erzeugen. Dazu stellt die PowerShell das CmdLet `New-Object` zur Verfügung. Als Parameter übergeben Sie den Namen der Klasse.

Mit

```
$Dlg=New-Object("System.Windows.Forms.OpenFileDialog")
```

erzeugen Sie daher eine Instanz der Klasse `OpenFileDialog` und speichern den Rückgabewert, also das erzeugte Objekt, in der Variablen `Dlg`.

[»] Damit Sie aus der Klasse ein Objekt erzeugen können, müssen Sie die entsprechende Objektbibliothek laden. Hier erledigt das schon die eingebundene Skript-Datei `wichtigefunktionen.ps1`. Möchten Sie die nicht laden, müssen Sie vor Aufruf des CmdLets `New`-Object die folgende Anweisung einfügen:

```
$erg=[reflection.assembly]::LoadWithPartialName(
   "System.Windows.Forms")
```

Haben Sie das Objekt erzeugt und gespeichert, können Sie das Dialogfeld mit `ShowDialog` anzeigen. Die Methode gibt einen Wert zurück, der festlegt, ob der Benutzer den Dialog mit der **Öffnen**- oder **Abbrechen**-Schaltfläche geöffnet hat. Hat der Benutzer auf **Öffnen** geklickt, gibt die Methode 1 zurück. Wenn Sie den ausgewählten Dateinamen ermitteln möchten, sollten Sie daher vorab prüfen, ob der Benutzer überhaupt auf **Öffnen** geklickt hat. Im Beispiel wird der Rückgabewert in der Variablen `Antw` gespeichert und abhängig von ihrem Wert danach der gewählte Dateiname ausgegeben, den Sie über die `FileName`-Eigenschaft ermitteln können.

```
#Skriptname: Dateiauswahl.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Verwendung des Dateiauswahl-Dialogs
#Anmerkungen: Benoetigt die Datei "wichtigefunktionen.ps1"
#Laden der Bibliotheksdateien

#Laden der Hilfsfunktionen
$bibpfad= Split-Path (Split-Path ($myInvocation.get_
MyCommand().Definition) -parent)  -parent
#$bibpfad="G:\GAL_powerShell\bsp"
. ($bibpfad + "\wichtigefunktionen.ps1")
#Benoetigte Variablen
$Dlg=""
$Antw=0
#Skriptbloecke und Funktionen

#Skriptinhalt
$Dlg=New-Object("System.Windows.Forms.OpenFileDialog")
$Antw=$Dlg.ShowDialog()
if ($Antw -eq 1)
{
   #Der Benutzer hat den Dialog mit OK geschlossen
   echo ($Dlg.Filename)
}
```

Rufen Sie das Skript auf, erscheint der folgende Dialog. Sie können hier alle möglichen Dateitypen auswählen und auf **Öffnen** klicken. Das Skript gibt dann die ausgewählte Datei mit Pfad aus:

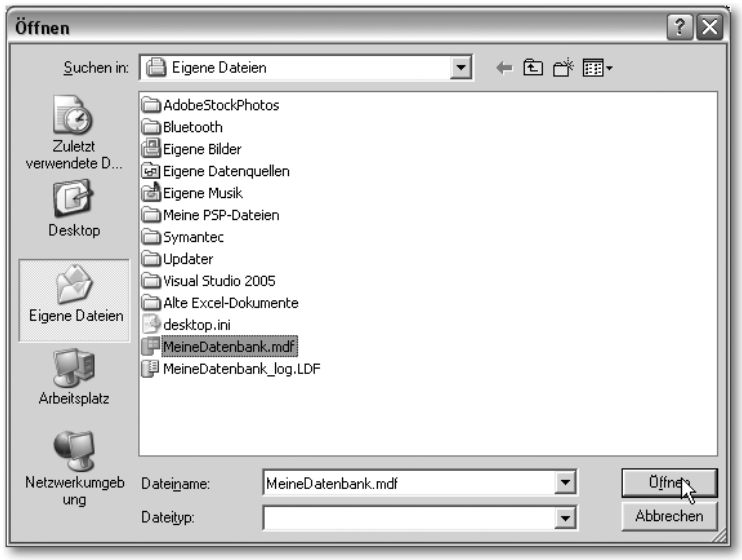

Abbildung 4.13 Der erzeugte Dateiauswahl-Dialog

Natürlich ist es zum einen nicht sonderlich sinnvoll, den Dateinamen nur auszugeben. Besser wäre, Sie erstellen sich eine Funktion, in der Sie den Dialog anzeigen und die den gewählten Dateinamen zurückgibt. Außerdem können Sie über die Eigenschaft `Filter` auch festlegen, welche Dateien der Benutzer auswählen können soll. Dazu definieren Sie einfach den Inhalt der Auswahlliste **Dateityp**.

Dazu müssten Sie das Skript wie folgt ändern. Zunächst deklarieren Sie dazu die Funktion und definieren dazu einen Parameter `strFilter`. Als Standardwert können Sie der Funktion den Filter für alle Dateien zuweisen. Dies ist die Zeichenkette `"Alle Dateien (*.*)|*.*"`. Jeder Dateifilter setzt sich aus zwei Teilen zusammen, die durch das Zeichen »|« getrennt werden. Der erste Teil ist der angezeigte Text, den Sie frei definieren können. Der zweite muss eine Filterbedingung darstellen. In der Regel wird dazu die Dateinamenserweiterung mit vorgestelltem Platzhalter verwendet. Zum Beispiel könnten Sie `*.mdb` angeben, wenn nur MDB-Dateien ausgewählt werden dürfen. Die Angabe `*.*` lässt daher die Auswahl aller Dateitypen zu.

Den Parameterwert weisen Sie dann der `Filter`-Eigenschaft zu, und zwar, nachdem Sie das Objekt erzeugt haben und bevor Sie die `ShowDialog`-Methode aufrufen. Nach Aufruf der Methode geben Sie dann die `FileName`-Eigenschaft aus der Funktion zurück.

```
#Skriptname: Dateiauswahl.ps1
#Autor: Helma Spona
```

```
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Verwendung des Dateiauswahl-Dialogs
#Anmerkungen: Benoetigt die Datei "wichtigefunktionen.ps1"
#Laden der Bibliotheksdateien

#Laden der Hilfsfunktionen
$bibpfad= Split-Path (Split-Path ($myInvocation.get_
MyCommand().Definition) -parent)  -parent
. ($bibpfad + "\wichtigefunktionen.ps1")

#Benoetigte Variablen
$Datei=""
$Dlg=""
$Antw=0
#Skriptbloecke und Funktionen
function Dateiauswahl([System.String]$strFilter="Alle Dateien
(*.*)|*.*")
{
   $Dlg=New-Object("System.Windows.Forms.OpenFileDialog")
   $Dlg.Filter=$strFilter
   $Antw=$Dlg.ShowDialog()
   if ($Antw -eq 1)
   {
      #Der Benutzer hat den Dialog mit OK geschlossen
      return $Dlg.Filename
   }
}
#Skriptinhalt
$Datei=DateiAuswahl
echo "Die Datei $Datei wurde gewaehlt!"
```

Wenn Sie das Skript nun aufrufen, wird die Funktion aufgerufen und das Dialogfeld wie vorher angezeigt, da auch jetzt alle Dateien ausgewählt werden können. Aber Sie haben nun die Möglichkeit, eine andere Filterbedingung zu definieren und an die Funktion zu übergeben.

Möchten Sie bspw. zwei Einträge in der Liste erzeugen, übergeben Sie einfach eine Filter-Angabe, die zwei Filterausdrücke enthält. Sie werden dann ebenfalls durch ein »|« getrennt.

Soll der Benutzer bspw. Textdateien und PowerShell-Skripte auswählen können, übergeben Sie als Filter-Zeichenfolge `"PowerShell-Skripten (*.ps1)|*.ps1|Text Dateien (*.txt)|*.txt"` und rufen damit die Funktion wie folgt auf:

```
$Datei=Dateiauswahl "PowerShell-Skripten (*.ps1)|*.ps1|Text-Dateien
(*.txt)|*.txt"
```

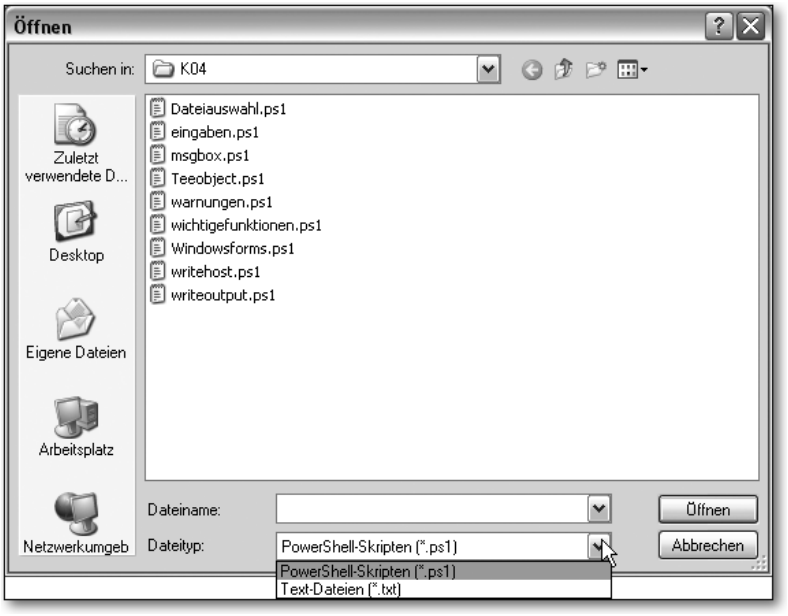

Abbildung 4.14 Die erzeugten Einträge in der Liste Dateityp

Sie können mit der Funktion nicht nur eine Datei auswählen lassen, sondern diese auch später bearbeiten und weiterverwenden. Wenn Sie den Benutzer bspw. eine PowerShell-Datei auswählen lassen, können Sie sie im Anschluss ausführen lassen. Falls der Benutzer wie hier noch die Möglichkeit hat, eine Textdatei auszuwählen, die nicht ausgeführt werden kann, müssen Sie allerdings prüfen, ob es sich um eine PowerShell-Datei handelt, indem Sie aus dem Dateinamen die Dateinamenserweiterung extrahieren. Dazu können Sie sich wieder eine Funktion erstellen, der Sie den Dateinamen übergeben und die die Erweiterung zurückgibt.

```
#Skriptname: Dateiauswahl.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Verwendung des Dateiauswahl-Dialogs
#Anmerkungen: Benoetigt die Datei "wichtigefunktionen.ps1"
#Laden der Bibliotheksdateien

#Laden der Hilfsfunktionen
...
```

```
#Skriptbloecke und Funktionen
...
function Dateityp ([System.String]$Dateiname)
{
   $Laenge=$Dateiname.Length
   if ($Laenge -gt 3)
   {
      $Laenge -=3
   }
   return $Dateiname.SubString($laenge,3)
}
#Skriptinhalt
...
```

Die einfachste Möglichkeit, die Dateinamenserweiterung zu ermitteln, besteht darin, die letzten drei Zeichen aus der Zeichenkette auszuschneiden. Dazu ermitteln Sie innerhalb der Funktion die Länge des Dateinamens und ziehen davon 3 ab. Die so berechnete Zahl übergeben Sie als ersten Parameter an die `SubString`-Methode. Der zweite Parameter gibt die Länge der zu ermittelnde Teilzeichenfolge an.

Problematisch ist diese Methode, wenn die Dateinamenserweiterung mehr oder weniger als drei Zeichen hat, bspw. ».JPEG« oder ».JS«. Besser ist daher, Sie nutzen für so etwas die passenden CmdLets und Objekte zur Verwaltung des Dateisystems. Mehr dazu erfahren Sie in Kapitel 5, *Arbeiten mit dem Dateisystem*. **[«]**

Wenn Sie nach Auswahl der Datei den Dateityp ermittelt haben, können Sie das ausgewählte Skript ausführen, indem Sie einfach ein »&« vor die Variable mit dem Dateinamen setzen:

```
...
$Datei=Dateiauswahl "PowerShell-Skripten (*.ps1)|*.ps1|Text-Dateien
(*.txt)|*.txt"
$Typ=Dateityp $Datei
echo "Ermittelter Dateityp: $Typ"
#Versuchen, die Datei auszufuehren
if ($Typ -eq "ps1")
{
   #Die Datei ausfuehren
   echo "Ausfuehren $Datei ...."
   &$Datei
}
...
```

4.3 Benutzeroberflächen gestalten

Neben den Dialogfeldern für Meldungen und die Dateiauswahl stellt das .NET-Framework noch weitere fertige Dialoge (bspw. für die Farbauswahl oder Druckerauswahl) zur Verfügung, die Sie analog nutzen können.

Dennoch werden Sie irgendwann an einen Punkt kommen, an dem einfache Dialoge nicht mehr reichen oder Sie ein Dialogfeld zur Eingabe von Werten erzeugen möchten, das analog zur `InputBox`-Funktion von VBA, VBScript oder Visual Basic funktioniert. Das alles geht ebenfalls mit Hilfe des .NET-Frameworks, indem Sie die Klasse `System.Windows.Forms.Form` verwenden.

Mit Hilfe dieser Klasse können Sie nicht nur einfache Dialoge erzeugen, sondern auch komplexe Benutzeroberflächen generieren. Die Grundlagen dazu erfahren Sie in den folgenden Abschnitten.

4.3.1 Einen Dialog erzeugen und anzeigen

Um mit Hilfe der Klasse `System.Windows.Forms.Form` einen Dialog zu erzeugen und anzuzeigen, müssen Sie zunächst wieder eine Instanz der Klasse erzeugen, indem Sie das CmdLet `New-Object` verwenden.

Damit erzeugen Sie ein neues Objekt aus der Klasse und weisen es der Variablen `Form` zu. Über diese Variable können Sie anschließend verschiedene Eigenschaften des Objekts festlegen, um das Aussehen zu beeinflussen.

Mit der Eigenschaft `TopMost` können Sie bspw. bestimmen, dass der Dialog alle anderen Fenster überdeckt und so auf jeden Fall sichtbar ist. Die `Text`-Eigenschaft legt den Fenstertitel fest. Um das Dialogfeld anzuzeigen, rufen Sie auch hier wieder die `ShowDialog`-Methode auf.

```
#Skriptname: NetObjekte.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K03
#Beschreibung: Erzeugen eines Dialogs mit der Klasse
#   System.Windows.Forms.Form
#Anmerkungen: keine

#Benoetigte Variablen
$Form=""
$Erg
#Skriptinhalt
$Erg=[reflection.assembly]::LoadWithPartialName(
   "System.Windows.Forms")
```

```
$Form=New-Object "System.Windows.Forms.Form"
$Form.TopMost = $true
$Form.Text="Meldung"
$Form.ShowDialog()
```

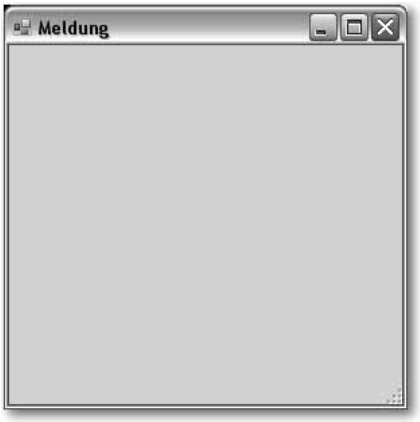

Abbildung 4.15 Der erzeugte Dialog

4.3.2 Steuerelemente einfügen und anordnen

So leer macht das Dialogfeld natürlich noch wenig Sinn. Sie müssen also noch Steuerelemente für das Formular erzeugen. Möchten Sie nur eine Meldung ausgeben, benötigen Sie ein Label-Steuerelement und eine Befehlsschaltfläche zum Schließen des Dialogfelds.

Steuerelemente erstellen Sie generell mit folgenden Schritten:

1. Ableiten des Objekts aus der Klasse des Steuerelements, bspw. `System.Windows.Forms.Label`
2. Festlegen der Steuerelement-Eigenschaften
3. Anfügen des Steuerelements an die `Controls`-Auflistung des Formulars

Folgendes Beispiel zeigt dies. Es fügt dem Formular einen Button und ein Label-Steuerelement hinzu. Label-Steuerelemente dienen zur Anzeige von Text. Sie leiten sie dazu aus der Klasse `System.Windows.Forms.Label` ab. Über die Eigenschaft `Text` legen Sie die Aufschrift für das Label-Feld fest. Mit der `Top`-Eigenschaft definieren Sie die Position des Steuerelements von oben. Diese bezieht sich auf das Fensterinnere, also ab da, wo der Inhalt des Fensters beginnt. Der Dialograhmen und der Fenstertitel gehören also nicht dazu.

Den Button erzeugen Sie aus der Klasse `System.Windows.Forms.Button` und setzen dann ebenfalls mit der `Text`-Eigenschaft seine Aufschrift und positionieren

ihn ausreichend tief unterhalb des Label-Feldes, damit er den Text nicht verdeckt.

Damit die beiden erstellten Steuerelemente auch in das Formular eingefügt und damit sichtbar werden, übergeben Sie sie an die `Add`-Methode der `Controls`-Auflistung. Damit sind die Steuerelemente eingefügt, und Sie können das Dialogfeld mit der `ShowDialog`-Methode aufrufen.

```
...
#Skriptinhalt
$Erg=[reflection.assembly]::LoadWithPartialName(
   "System.Windows.Forms")
$form=New-Object "System.Windows.Forms.Form"
$form.TopMost = $true
$form.Text="Meldung"

$label=New-Object "System.Windows.Forms.Label"
$label.Text='Hallo Welt'
$label.Top=10

$button = New-Object "System.Windows.Forms.Button"
$button.Text = "OK"
$button.Top=100

$form.Controls.Add($label)
$form.Controls.Add($button)
$form.ShowDialog()
...
```

Optimieren lässt sich das Ganze noch, indem Sie die Position der Schaltfläche von der Höhe und der Position des Label-Feldes abhängig berechnen und abhängig davon auch die Höhe des Formulars berechnen. Dazu müssten Sie folgende Änderungen am Code vornehmen:

```
$label=New-Object "System.Windows.Forms.Label"
$label.Text='Hallo Welt'
$label.Height=50
$label.Top=10

$button = New-Object "System.Windows.Forms.Button"
$button.Text = "OK"
$button.Add_Click({$form.Dispose()})
$button.Top=$label.top + $label.height + 10

$form.Controls.Add($label)
```

```
$form.Controls.Add($button)

$form.Height=$label.top + $label.height + 10 +
   $button.height + $button.top
$form.ShowDialog()
```

Abbildung 4.16 Das erzeugte Fenster mit der Meldung und dem Button

4.3.3 EventHandler für Buttons erstellen

Allerdings wird das Formular nun noch nicht geschlossen, wenn der Benutzer auf den Button klickt. Um das zu erreichen, müssen Sie dem Formular einen EventHandler für das `Click`-Ereignis zuweisen.

> Ein EventHandler ist eine spezielle Prozedur, die ausgeführt wird, wenn das zugeordnete Ereignis eintritt. Das `Click`-Ereignis tritt ein, wenn der Benutzer mit der Maus auf den Button klickt. **[«]**

Anders als in anderen Programmiersprachen müssen Sie dazu in der PowerShell aber keine Prozedur erstellen, sondern Sie übergeben an die Methode `Add_Click` einen Skriptblock, der ausgeführt wird, wenn das Ereignis eintritt. Möchten Sie bspw. die `Dispose`-Methode des Dialogs aufrufen, müssen Sie dazu folgenden Code einfügen:

```
$button.Add_Click({$form.Dispose()})
```

Die `Dispose`-Methode führt dazu, dass das Objekt aus dem Speicher entfernt wird, in diesem Fall also das Formular. Dies ist allerdings eine sehr abrupte Art und Weise, das Formular zu schließen. Besser ist, Sie rufen zuvor die `Close`-Methode auf. Wenn Sie zwei Anweisungen nacheinander angeben möchten, trennen Sie diese mit einem Semikolon:

```
...
$button = New-Object "System.Windows.Forms.Button"
$button.Text = "OK"
$button.Add_Click({$form.Close();$form.Dispose()})
$button.Top=$label.top + $label.height + 10
...
```

Rufen Sie nun das Formular auf, können Sie es auch mit einem Klick auf den Button öffnen.

4.3.4 Eine InputBox-Funktion für Benutzereingaben programmieren

Benutzereingaben können Sie in einer grafischen Dialogbox nur anfordern, indem Sie selbst ein Eingabeformular erzeugen und anzeigen. Dazu nehmen Sie die Klasse `System.Windows.Forms.Form` zu Hilfe.

Sie müssen dazu eine Form anzeigen, die einen **Abbrechen**- und einen **OK**-Button hat und darüber hinaus über ein Eingabefeld und ein Label-Feld verfügt.

Das Eingabefeld benötigen Sie, damit der Benutzer dort die geforderte Eingabe vornehmen kann. Mit dem Label-Feld können Sie den Text anzeigen, der den Benutzer zur Eingabe auffordert. Über die beiden Schaltflächen kann der Benutzer das Formular schließen und entweder den Wert `false` oder den eingegebenen Wert zurückgeben.

Optimal ist eine Funktion, die den Eingabedialog anzeigt und den eingegebenen Wert zurückgibt. An die Funktion übergeben Sie drei Werte, den Titel des Dialogs, den Text für das Label-Steuerelement und den Wert, der als Vorschlag im Eingabefeld angezeigt werden soll. Die Deklaration der Funktion muss also wie folgt lauten:

```
function inputbox([System.String]$strPrompt="",
   [System.String]$strTitel="", [System.String]$strWert="")
{
   ...
}
```

Innerhalb der Funktion deklarieren Sie zunächst die benötigten Variablen. `form` speichert das `System.Windows.Forms.Form`-Objekt, `erg` speichert den Rückgabewert der `ShowDialog`-Methode, und `eingabe` speichert den Text des Eingabefeldes beim Schließen der Form. Dies ist also der Rückgabewert des Dialogfeldes, nicht jedoch zwingend der Rückgabewert der Funktion.

Als Erstes erzeugen Sie dann das Formular und setzen dessen `TopMost`-Eigenschaft auf `true`, damit das Formular im Vordergrund angezeigt wird. Anschließend setzen Sie dessen `Text`-Eigenschaft auf den Parameter `strTitel`.

```
#Skriptname: inputbox.ps1
#Autor: Helma Spona
#Auflage: 1
#Verzeichnis: /Bsp/K04
#Beschreibung: Zeigt die Erstellung von Eingabedialogen
```

```
#Anmerkungen: Benoetigt die Datei "wichtigefunktionen.ps1"
#Laden der Bibliotheksdateien

#Laden der Hilfsfunktionen
$bibpfad= Split-Path (Split-Path ($myInvocation.get_
MyCommand().Definition) -parent)  -parent
#$bibpfad="G:\GAL_powerShell\bsp"
. ($bibpfad + "\wichtigefunktionen.ps1")

#Skriptbloecke und Funktionen
function inputbox([System.String]$strPrompt="",
[System.String]$strTitel="", [System.String]$strWert="")
{
   $form=""
   $erg=$false
   $eingabe=""

   $form=New-Object "System.Windows.Forms.Form"
   $form.TopMost = $true
   $form.Text=$strTitel
...
```

Als Nächstes fügen Sie die die benötigten Steuerelemente ein: zuerst das Label-Feld.

[+] Sie sollten die Steuerelemente immer in der Reihenfolge einfügen, in der die Steuerelemente auch im Formular angezeigt werden sollen, weil Sie dann die Position der Steuerelemente anhand der Positionen und Größen der vorangegangenen Steuerelemente berechnen können.

Die Aufschrift des Label-Steuerelements legen Sie mit dem Parameter `strPrompt` fest. Damit etwas Abstand zur oberen Innenkante des Fensters vorhanden ist, bekommt die `Top`-Eigenschaft den Wert 10. Die Breite wird auf die Fensterbreite abzüglich 10 festgelegt, um auch hier einen kleinen Abstand zu schaffen. Wenn Sie dann die Hälfte dieses Abstandes der `Left`-Eigenschaft zuweisen, steht das Steuerelement mittig innerhalb des Formulars.

```
...
   $label=New-Object "System.Windows.Forms.Label"
   $label.Height=20
   $label.Text=$strPrompt
   $label.Top=10
   $label.Width=$form.width-10
   $label.Left=5
...
```

Das Eingabefeld erzeugen Sie anschließend aus der Klasse `System.Windows.Forms.Textbox`. Der Wert, der im Textfeld angezeigt wird, steht in der `Text`-Eigenschaft. Sie können also den übermittelten Anfangswert der `Text`-Eigenschaft zuweisen und auch später den eingegebenen Wert über die `Text`-Eigenschaft abrufen. Mit `Top` und `Left` legen Sie die Position des Steuerelements fest.

```
...
   #Eingabefeld erzeugen
   $eingabefeld=New-Object "System.Windows.Forms.Textbox"
   $eingabefeld.Height=20
   $eingabefeld.Text=$strWert
   $eingabefeld.Top=$label.top +$label.height+5
   $eingabefeld.Left=$label.Left
...
```

Im Unterschied zum vorherigen Beispiel benötigen Sie nun zwei Schaltflächen, eine **OK**- und eine **Abbrechen**-Schaltfläche. Für jede dieser Schaltflächen erstellen Sie einen EventHandler mit der `Add_Click`-Methode. Allerdings verwenden beide unterschiedlichen Code. Beim Klicken auf **OK** soll der eingegebene Text zurückgegeben werden. Zudem soll auch ermittelt werden können, dass der Benutzer auf **OK** und nicht auf **Abbrechen** geklickt hat.

Das heißt, Sie müssen eigentlich zwei Werte zurückgeben. Das funktioniert, indem Sie die Werte in entsprechenden Variablen speichern. Der EventHandler für den **OK**-Button setzt zunächst die Variable `erg` auf `true` und weist dann der Variablen `eingabe` den Inhalt des Textfeldes zu. Danach wird das Formular geschlossen.

Der Code für den **Abbrechen**-Button setzt die Variable `erg` auf `false`. Anschließend werden alle Steuerelemente mit der `Add`-Methode an die `Controls`-Auflistung angehängt. Anschließend wird noch die Höhe des Formulars berechnet und dann der Dialog mit der `ShowDialog`-Methode angezeigt.

```
...
   #Buttons erstellen
   $bttOK= New-Object "System.Windows.Forms.Button"
   $bttOK.Text = "OK"
   $bttOK.Add_Click({$erg=$true;
      $eingabe=$eingabefeld.Text; $form.Close();
      $form.Dispose()})
   $bttOK.Top=$eingabefeld.top + $eingabefeld.height + 10
   $bttOK.Width=70
   $bttOK.Left=$label.Left
   $bttAbbrechen= New-Object "System.Windows.Forms.Button"
```

```
   $bttAbbrechen.Text = "Abbrechen"
   $bttAbbrechen.Add_Click({$erg=$false;$eingabe="";
      $form.Close();$form.Dispose();})
   $bttAbbrechen.Top=$bttOK.top
   $bttAbbrechen.Width=70
   $bttAbbrechen.Left=$bttOK.left + $bttOK.width + 10

   $form.Controls.Add($label)
   $form.Controls.Add($bttOK)
   $form.Controls.Add($bttAbbrechen)
   $form.Controls.Add($eingabefeld)
   $form.Height=$eingabefeld.top + $eingabefeld.height +
       10 + $bttOK.height + $bttOK.top
   $temp=$form.ShowDialog()
...
```

Der Code nach Aufruf der `ShowDialog`-Methode wird erst ausgeführt, wenn das Dialogfeld wieder geschlossen wurde. Das heißt, dann haben die beiden Variablen `erg` und `eingabe` ihre Werte, und Sie können diese abrufen und prüfen, um den Rückgabewert der Funktion zu definieren. Sie müssen dazu nur abfragen, ob die Variable `erg` den Wert `true` hat. Falls ja, geben Sie den Wert der Variablen `eingabe` zurück, ansonsten `false`.

```
...
   if ($erg -eq $true)
   {
      return $eingabe
   }
   else
   {
      return $false
   }

}
...
```

Sie können nun die Funktion aufrufen und so eine grafische Eingabeaufforderung anzeigen.

```
...
#Skriptinhalt

$Erg=[reflection.assembly]::LoadWithPartialName(
   "System.Windows.Forms")

echo (inputbox "Bitte Wert eingeben!" "Eingabe notwendig")
```

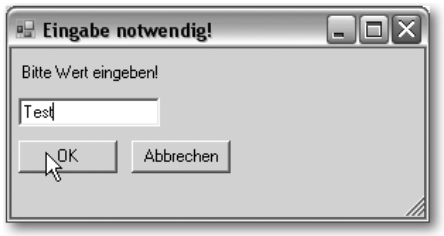

Abbildung 4.17 Die angezeigte Eingabeaufforderung

4.3.5 Aktives Steuerelement und Tabulatorreihenfolge festlegen

Selbstverständlich können Sie das Dialogfeld auch noch formatieren und bspw. Farben für Aufschriften festlegen oder das aktive Steuerelement bestimmen. Auch EventHandler für andere Ereignisse sind durchaus denkbar. Einige Möglichkeiten sollen abschließend noch gezeigt werden.

Das erste, sehr lästige Problem des Eingabedialogs ist, dass der Benutzer nicht nur den Dialog aktivieren muss, bevor er eine Eingabe vornehmen kann. Er muss dann auch explizit den Cursor in das Eingabefeld setzen. Das können Sie vermeiden, indem Sie für die einzelnen Steuerelemente die Tabulatorreihenfolge setzen.

Die Tabulatorreihenfolge ist die Reihenfolge, in der die Steuerelemente aktiviert werden. Das Steuerelement mit dem Index 0 wird automatisch aktiviert, wenn das Formular angezeigt wird.

Die Tabulatorreihenfolge können Sie festlegen, indem Sie für alle anklickbaren oder aktivierbaren Steuerelemente die `TabIndex`-Eigenschaft setzen. Damit das Eingabefeld automatisch aktiviert ist, müssen Sie ihm den Wert 0 geben.

```
function inputbox([System.String]$strPrompt="",
   [System.String]$strTitel="", [System.String]$strWert="")
{
   $form=""
   $erg=$false
   $eingabe=""
   $form=New-Object "System.Windows.Forms.Form"
   $form.TopMost = $true
   $form.Text=$strTitel
   $label=New-Object "System.Windows.Forms.Label"
   $label.Height=20
   $label.Text=$strPrompt
   $label.Top=10
   $label.Width=$form.width-10
```

```
   $label.Left=5
   #Eingabefeld erzeugen
   $eingabefeld=New-Object "System.Windows.Forms.Textbox"
   $eingabefeld.Height=20
   $eingabefeld.Text=$strWert
   $eingabefeld.Top=$label.Top +$label.Height+5
   $eingabefeld.Left=$label.Left
   $eingabefeld.TabIndex=0
   #Buttons erstellen
   $bttOK= New-Object "System.Windows.Forms.Button"
   $bttOK.Text = "OK"
      $bttOK.Add_Click({$erg=$true;
      $eingabe=$eingabefeld.Text; $form.Close();
      $form.Dispose()})
   $bttOK.Left=$label.Left
   $bttOK.Top=$eingabefeld.top + $eingabefeld.height + 10
   $bttOK.Width=70
   $bttOK.TabIndex=1
   $bttAbbrechen= New-Object "System.Windows.Forms.Button"
   $bttAbbrechen.Text = "Abbrechen"
   $bttAbbrechen.Add_Click({$erg=$false;$eingabe="";
      $form.Close();$form.Dispose();})
   $bttAbbrechen.Top=$bttOK.top
   $bttAbbrechen.Width=70
   $bttAbbrechen.Left=$bttOK.left + $bttOK.width + 10
   $bttAbbrechen.TabIndex=2
   $form.Controls.Add($label)
   $form.Controls.Add($bttOK)
   $form.Controls.Add($bttAbbrechen)
   $form.Controls.Add($eingabefeld)
   #$eingabefeld.SetFocus
   $form.Height=$eingabefeld.top + $eingabefeld.height +
      10 + $bttOK.height + $bttOK.top
   $temp=$form.ShowDialog()
   if ($erg -eq $true)
   {
      return $eingabe
   }
   else
   {
      return $false
   }

}
```

Nun muss der Benutzer nur noch den Dialog anklicken, um ihn zu aktivieren, und kann dann mit der Eingabe beginnen, da der Cursor automatisch im Eingabefeld steht. Drückt er danach ⇆, wird erst der OK-Button und beim zweiten ⇆ der **Abbrechen**-Button aktiviert.

4.3.6 Farben ändern

Wenn Sie einem Steuerelement eine Farbe zuweisen möchten, benötigen Sie dazu ein `Color`-Objekt. Jedes `Color`-Objekt stellt eine Farbe dar und stellt Methoden bereit, die eine Konvertierung und Änderung der Farbe ermöglichen.

`Color`-Objekte erzeugen Sie aus der Klasse `System.Drawing.Color`. Da sie nicht Bestandteil des Namensraums `System.Windows.Forms` ist, müssen Sie auch diese Ressource zunächst laden und daher vor dem Aufruf der Funktion `inputbox` die folgende Anweisung einfügen:

```
$Erg=[reflection.assembly]::LoadWithPartialName(
   "System.Drawing")
```

Dann können Sie die statische Methode `FromArgb` nutzen, um eine RGB-Farbe zu erstellen. Ihr übergeben Sie nacheinander den Farbanteil für Rot, Grün und Blau. Für alle drei Farben können Sie Werte von 0 bis 255 einschließlich eingeben. Gleiche Werte für alle drei Parameter stellen einen Grauton dar. Geben Sie für alle Werte 0 an, stellt dies die Farbe Schwarz dar. Geben Sie für alle Farben 255 an, ergibt sich die Farbe Weiß. Die Methode gibt ein `Color`-Objekt zurück, das die entsprechende Farbe repräsentiert.

Wenn Sie das Label-Feld mit roter Schrift versehen möchten, müssen Sie dazu den Code wie folgt ergänzen:

```
function inputbox([System.String]$strPrompt="",
[System.String]$strTitel="", [System.String]$strWert="")
{
   $form=""
   $erg=$false
   $eingabe=""
   $farbe=0
   $farbe=[System.Drawing.Color]::FromArgb(255, 0, 0)
   $form=New-Object "System.Windows.Forms.Form"
   $form.TopMost = $true
   $form.Text=$strTitel

   $label=New-Object "System.Windows.Forms.Label"
   $label.Height=20
   $label.Text=$strPrompt
```

```
   $label.Top=10
   $label.Width=$form.width-10
   $label.Left=5
   $label.ForeColor=$farbe
...
```

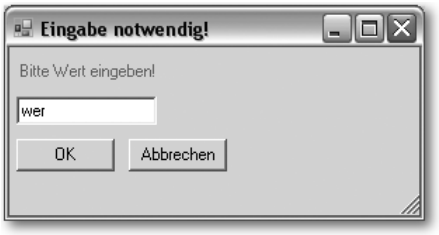

Abbildung 4.18 Das Ergebnis: Die Eingabeaufforderung in roter Schrift

Sie können aber über die `ForeColor`-Eigenschaft nicht nur die Schriftfarbe für ein Steuerelement festlegen, sondern mit der `BackColor`-Eigenschaft auch die Hintergrundfarbe. Die können Sie sowohl für das Formular selbst als auch für die Steuerelemente festlegen und so auch ganz farbenfrohe Dialoge gestalten. Die folgende Änderung am Code bewirkt, dass das Formular eine gelbe Hintergrundfarbe bekommt und die Schaltflächen orange formatiert werden.

[+] Die Farbe Gelb erzeugen Sie, indem Sie den Rot- und Grün-Anteil der Farbe auf 255 setzen. Für die Farbe Orange reduzieren Sie den Grün-Anteil gegenüber Gelb um die Hälfte.

Wenn Sie Probleme haben, die korrekten Farbwerte für die Zielfarbe zu finden, nehmen Sie ein Grafikprogramm wie Photoshop oder Paint Shop Pro zur Hand. Dort können Sie im Farbauswahl-Dialog eine Farbe wählen, und das Programm zeigt die einzelnen Farbanteile an. Diese müssen Sie dann nur noch im Code eintragen.

Der zu ändernde Code sieht wie folgt aus:

```
...
   $bttOK.Left=$label.Left
   $bttOK.Top=$eingabefeld.top + $eingabefeld.height + 10
   $bttOK.Width=70
   $bttOK.TabIndex=1
   $bttAbbrechen= New-Object "System.Windows.Forms.Button"
   $bttAbbrechen.Text = "Abbrechen"
   $bttAbbrechen.Add_Click({$erg=$false;$eingabe="";
      $form.Close();$form.Dispose();})
   $bttAbbrechen.Top=$bttOK.top
   $bttAbbrechen.Width=70
   $bttAbbrechen.Left=$bttOK.left + $bttOK.width + 10
```

```
   $bttAbbrechen.TabIndex=2
   $farbe=[System.Drawing.Color]::FromArgb(255, 128, 0)
   $bttOK.BackColor=$farbe
   $bttAbbrechen.BackColor=$farbe
   $form.Backcolor=[System.Drawing.Color]::FromArgb(
      255, 255, 0)
...
```

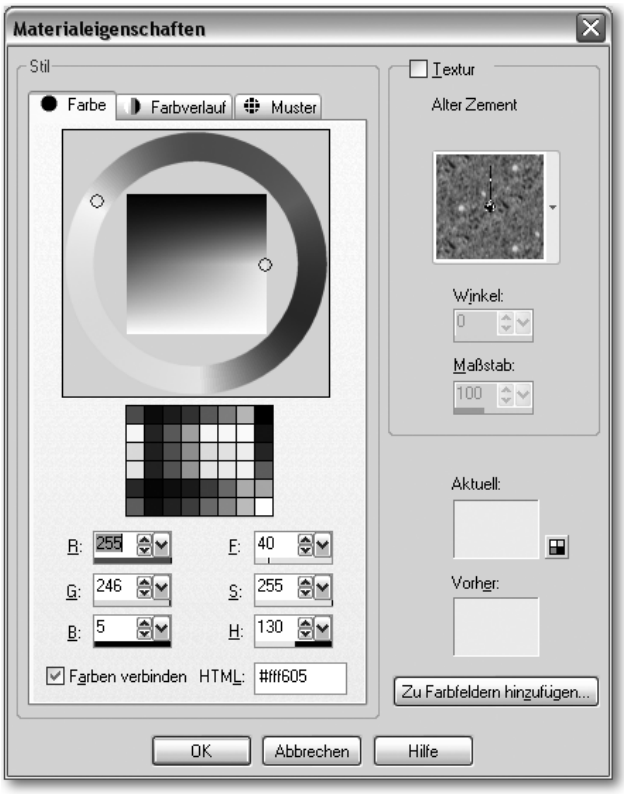

Abbildung 4.19 Ermitteln der Farbanteile für die Zielfarbe, hier mit Paint Shop Pro X

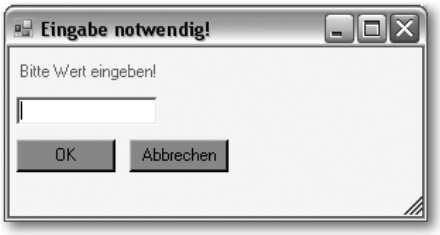

Abbildung 4.20 Das eingefärbte Dialogfeld

4.3.7 EventHandler erstellen

EventHandler sind ein wichtiger Bestandteil von Benutzeroberflächen, weil Sie damit auf Benutzeraktionen reagieren können. Sie können diese aber nicht nur für das `Click`-Ereignis der Buttons erstellen, sondern für viele weitere Ereignisse. In jedem Fall erzeugen Sie den EventHandler mit einer Methode, deren Name sich aus »`Add_`«, gefolgt vom Namen des Ereignisses zusammensetzt.

Möchten Sie bspw. Code definieren, der beim Laden des Formulars ausgeführt wird, erstellen Sie dazu einen EventHandler für das `Load`-Ereignis.

Das `Load`-Ereignis tritt ein, wenn das Formular geladen wird. Das ist nicht dann der Fall, wenn Sie das `System.Windows.Forms.Form`-Objekt erzeugen, sondern dann, wenn Sie dessen `ShowDialog`-Methode aufrufen.

Sie können sich dieses Ereignis bpsw. zunutze machen, um abhängig von den übergebenen Parameterwerten der Funktion zu prüfen, ob es Sinn macht, den **OK**-Button zu aktivieren. Auf diese Weise können Sie die Benutzer zu einer gültigen Eingabe zwingen. Sie müssen dazu nur beim Laden des Formulars prüfen, ob der Wert des Eingabefeldes größer als eine leere Zeichenfolge ist. Falls ja, aktivieren Sie den Button über die `Enabled`-Eigenschaft, oder Sie deaktivieren ihn, indem Sie die Eigenschaft auf `false` setzen.

Um einen solchen EventHandler zu definieren, müssen Sie den Code wie folgt ergänzen. Sie sehen daran schon, dass Sie auch den Code für einen EventHandler mehrzeilig definieren können.

Damit ein Leerzeichen nicht als Eingabe akzeptiert wird, sollten Sie die `Trim`-Methode verwenden, um vor dem Vergleich führende und abschließende Leerzeichen zu entfernen. **[+]**

```
...
   $form.Height=$eingabefeld.top + $eingabefeld.height + 10
      + $bttOK.height + $bttOK.top
   $form.Add_Load({
      if ($eingabefeld.Text.Trim() -gt "")
      {
         $bttOK.Enabled=$true
      }
      else
      {
         $bttOK.Enabled=$false
      }
   })
   $temp=$form.ShowDialog()
...
```

Der vorstehende Code prüft im Prinzip nur, ob als dritter Parameter ein Wert ungleich einer leeren Zeichenfolge übergeben wurde. Der Button wird nun aber nicht automatisch aktualisiert, wenn der Benutzer Eingaben vornimmt. Dazu benötigen Sie einen zweiten EventHandler für das `TextChanged`-Ereignis des Eingabefeldes. Das Ereignis tritt ein, wenn der Benutzer den Inhalt des Eingabefeldes geändert hat. Für dieses Ereignis erstellen Sie einen EventHandler mit dem gleichen Code:

```
...
   $eingabefeld.Add_TextChanged({
      if ($eingabefeld.Text.Trim() -gt "")
      {
         $bttOK.Enabled=$true
      }
      else
      {
         $bttOK.Enabled=$false
      }
   })
   $temp=$form.ShowDialog()
...
```

Wenn Sie nun die Funktion aufrufen und als dritten Parameter nichts oder eine leere Zeichenfolge angeben, ist der **OK**-Button deaktiviert.

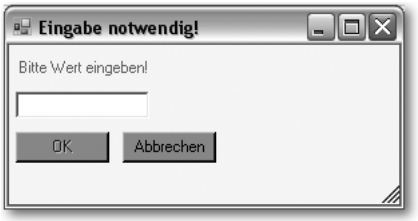

Abbildung 4.21 Bei der Anzeige des Dialogs ist die Schaltfläche deaktiviert.

Wenn Sie nun aber mindestens ein Zeichen eingeben, das kein Leerzeichen ist, wird die Schaltfläche durch den EventHandler für das `TextChanged`-Ereignis automatisch aktiviert und auch wieder deaktiviert, wenn Sie den Inhalt des Textfeldes löschen.

[+] Statt den Button über die `Enabled`-Eigenschaft zu deaktivieren, können Sie ihn über die `Visible`-Eigenschaft auch ein- (`true`) bzw. ausblenden (`false`).

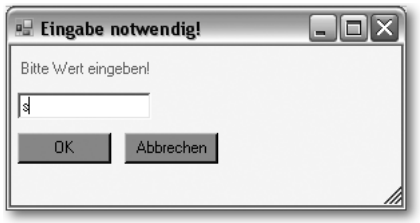

Abbildung 4.22 Nach der ersten Eingabe ist der Button aktiviert.

Es ist natürlich nicht sehr wartungsfreundlich, wenn Sie zwei EventHandler mit dem gleichen Code haben und den Code zweimal eingeben und dann auch zweimal warten müssen. Um das zu vermeiden, können Sie auch den Inhalt der EventHandler vorab als Codeblock definieren und in den EventHandlern nur aufrufen. Im folgenden Listing wird der Codeblock der Variablen `pruefen` zugewiesen und diese im EventHandler aufgerufen.

```
function inputbox([System.String]$strPrompt="",
[System.String]$strTitel="", [System.String]$strWert="")
{
   $form=""
   $erg=$false
   $eingabe=""
   $farbe=0
   $pruefen={
      if ($eingabefeld.Text.Trim() -gt "")
      {
         $bttOK.Enabled=$true
      }
      else
      {
         $bttOK.Enabled=$false
      }
   }
   $farbe=[System.Drawing.Color]::FromArgb(255, 0, 0)
   $form=New-Object "System.Windows.Forms.Form"
...
   $form.Height=$eingabefeld.top + $eingabefeld.height +
      10 + $bttOK.height + $bttOK.top
   #EventHandler erstellen
   $bttAbbrechen.Add_Click({$erg=$false;$eingabe="";
      $form.Close();$form.Dispose();})
   $bttOK.Add_Click({$erg=$true;
      $eingabe=$eingabefeld.Text; $form.Close();
```

```
      $form.Dispose()})
   $form.Add_Load({&$pruefen})
   $eingabefeld.Add_TextChanged({&$pruefen})
   $temp=$form.ShowDialog()
   #Rueckgabewert pruefen
   if ($erg -eq $true)
   {
      return $eingabe
   }
   else
   {
      return $false
   }

}
```

[+] Wenn Sie die Funktion `InputBox` auch in das Skript `wichtigeFunktionen.ps1` einfügen, können Sie mit ihrer Hilfe ganz einfach in vielen Skripten Eingabeaufforderungen anzeigen.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z