11.5.0.1 Fehlerbehandlung 2

Zum Fehler-Management zählen auch in der Programmiersprache Gambas die Themen Fehler erkennen, Fehler abfangen und Fehler behandeln. Gambas stellt dafür neben den Instruktionen TRY, FINALLY, CATCH, ERROR, DEBUG auch die Fehlerklasse Error, eine globale, nur lesbare boolsche Variable Error und Fehlernummern sowie Fehlerbeschreibungen zur Verfügung, die in diesem Kapitel beschrieben werden.

11.5.0.1.1 Fehler-Klasse Error (gb)

Die Fehler-Klasse verfügt diese statischen, nur lesbaren Eigenschaften:

Eigenschaft	Datentyp	Beschreibung
Backtrace	String[]	Gibt eine Rückverfolgung des Stack-Zustandes des Funktionsaufrufs zurück, als der letzte Fehler aufgetreten ist. Der erste Eintrag des String-Arrays ist der tiefste Funktionsaufruf. Wenn keine Debugging-Informationen verfügbar waren oder wenn kein Fehler aufgetreten ist, gibt diese Eigenschaft NULL zurück.
Class	Class	Liefert den Namen der Klasse, in der der letzte Fehler aufgetreten ist.
Code	Integer	Gib die letzte Fehler-Nummer zurück.
Text	String	Gibt die letzte Fehler-Meldung - korrespondierend zum dazugehörenden Fehler-Code in englischer Sprache zurück.
Where	String	Gibt eine Zeichenkette zurück, die die Position des letzten Fehlers im Quellcode beschreibt.

Tabelle 11.5.0.1.1 : Eigenschaften der Fehler-Klasse

Statt der Eigenschaft Error.Backtrace der Klasse Error können Sie auch die statische Eigenschaft System.Backtrace (gb) mit Static Property Read Backtrace As String[ ] verwenden. In beiden Fällen wird Ihnen ein String-Array zurückgegeben, das Sie auslesen und anzeigen lassen können.

Die Fehler-Klasse Error besitzt nur drei Methoden:

Methode	Beschreibung
Clear()	Setzt den Fehler-Code auf 0 und die Fehler-Meldung auf NULL. Die Methode können Sie verwenden, wenn Sie innerhalb einer Prozedur den Fehler behandeln und danach die Eigenschaften Error.Code auf 0 und Error.Text auf NULL setzen.
Propagate()	Die Methode sorgt dafür dass der Fehler erneut ausgelöst wird.
Raise(argMessage As String)	Löst einen vom Anwender definierten Fehler aus. Die Fehlermeldung in *argMessage* können Sie im Quelltext frei festlegen.

Tabelle 11.5.0.1.2 : Methoden der Fehler-Klasse

11.5.0.1.2 Anweisung DEBUG

```
DEBUG Expression [ { ; | ;; | , } Expression ... ] [ { ; | ;; | , }  ]
```

- Die Anweisung gibt eine Liste von Ausdrücken auf die Standardfehlerausgabe aus. Aber nur dann, wenn das Programm mit Debugging-Informationen kompiliert wurde. Die Einstellung können Sie im Dialog "Ausführbare Datei erstellen" ändern. Standard ist die Festlegung "Debugging-Informationen in der ausführbaren Datei behalten".
- Die Position der DEBUG-Anweisung im Quelltext wird Allem vorangestellt.
- Die Ausdrücke werden mit der Str$-Funktion in Zeichenketten umgewandelt, welche die aktuellen Standortparameter wie Sprache der GUI, Zahlen-, Währungs-, Datums- und Zeitformate, Zeichensatz und Tastatur-Layout (Locale) berücksichtigt!
- Steht nach dem letzten Ausdruck kein Semikolon und kein Komma, so wird nach dem letzten Ausdruck ein Zeilenumbruch-Zeichen ausgegeben.
- Wenn das Semikolon verdoppelt wird, so wird zwischen den Ausdrücken jeweils ein Leerzeichen ausgegeben.
- Wird anstelle eines Semikolons ein Komma verwendet, so wird zur Trennung der Ausdrücke jeweils ein Tabulatorzeichen (ASCII-Code 9, gb.Tab) ausgegeben.

Der Liste von Ausdrücken wird der Name der Klasse, ein Punkt, der Name der Prozedur, ein Doppelpunkt und die Zeilennummer vorangestellt wie zum Beispiel:

```
FMain.GetStart.69: /tmp/gambas.1000/13839/New.tmp/new.md.
```

Im folgenden Beispiel ging es darum, diverse Pfade zu kontrollieren. Die Anweisung DEBUG – zugeschaltet durch die boolsche Variable bDebug – wurde einer Anweisung PRINT vorgezogen, da auch die Zeilen-Nummern im Quelltext interessierten:

```
  Public bDebug As Boolean

  bDebug = True

  sBasicConfigDir = Desktop.ConfigDir &/ sVendor &/ Lower(sAppName)

  If bDebug Then Debug sBasicConfigDir

  If Not Exist(sBasicConfigDir) Then Shell.MkDir(sBasicConfigDir)
  hSettings = New Settings(sBasicConfigDir &/ File.SetExt(Lower(sAppName), "conf"))

  If bDebug Then Debug sBasicConfigDir &/ File.SetExt(Lower(sAppName), "conf")
  ...
  sCurrentMDFilePath = sTempDir &/ "new.md"

  If bDebug Then Debug sCurrentMDFilePath
```

Ausgaben in der Konsole der IDE:

```
FMain._new.49: /home/hans/.config/gambasbook/mdeditor
FMain._new.52: /home/hans/.config/gambasbook/mdeditor/mdeditor.conf
...
FMain.GetStart.69: /tmp/gambas.1000/13839/New.tmp/new.md
```

11.5.0.1.3 Anweisung ERROR

```
ERROR Expression [ { ; | ;; | , } Expression ... ] [ { ; | ;; | , }  ]
```

Die Anweisung gibt eine Liste von Ausdrücken auf die Standardfehlerausgabe aus – genau wie die Anweisung PRINT. Wenn Sie ein Kommandozeilenprogramm schreiben, dann wäre diese Anweisung die nicht-graphische Entsprechung von Message.Error().

Wenn Sie in der IDE im Menü Debuggen die Einträge 'Terminalemulator benutzen' und "Standardfehlerausgabe umleiten" beide mit ✔ aktivieren, dann werden DEBUG- und ERROR-Ausgaben in der IDE-Konsole angezeigt und PRINT-Ausgaben im Terminal. Wenn aber 'Terminalemulator benutzen' aktiv ist und die Umleitung deaktiviert wurde, so werden DEBUG-, ERROR- und PRINT-Ausgaben im Terminal angezeigt:

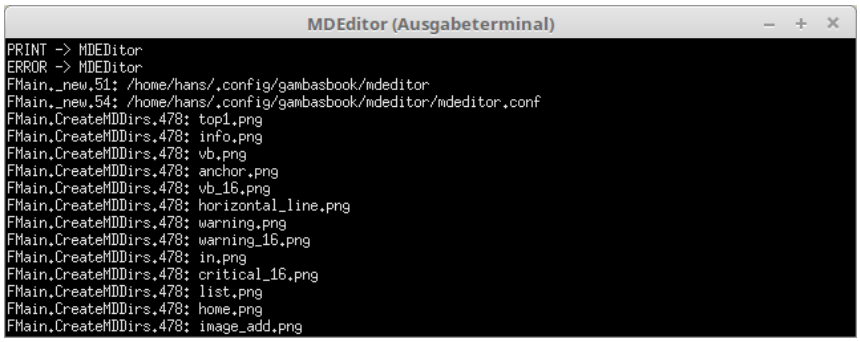

Abbildung 11.5.0.1.1: DEBUG-, ERROR- und PRINT-Ausgaben im (IDE-)Terminal

Die Standardausgabe kann durch die Anweisung ERROR TO umgeleitet werden (▷ Kapitel 6.2.0.14 ERROR TO DEFAULT). Eine Umleitung ist einfach realisiert:

```
DIM aFile As File

  aFile = OPEN User.Home &/ "error_ping.log" FOR CREATE ' or APPEND

  OUTPUT TO aFile
  ...
```

```
    Error ...
  ...
  CLOSE aFile
```

Dieser Quelltext-Ausschnitt leitet die Ausgaben der Error-Anweisung in die angegebene Datei um.

11.5.0.1.4 Globale Variable Error

Die boolsche Variable Error – global und nur lesbar – liefert den Wert True, wenn ein Fehler aufgetreten ist. Verwenden Sie es direkt nach einer Try-Anweisung, um zu erfahren, ob die dort angegebene Anweisung fehlgeschlagen ist oder nicht. Um weitere Informationen über den Fehler zu erhalten, setzen Sie mit Erfolg die (Fehler-)Klasse Error (gb) ein.

Beispiel: Löschen einer Datei

```
Public Sub btnKillFile_Click()

  Dim sMessage, sFilePath As String

  sFilePath = User.Home &/ "Temp" &/ "tmp.text"

  Try Kill sFilePath
  If Error Then Print "Error! The file `" & File.Name(sFilePath) & "` cannot be deleted!"
  If Error Then Error Subst("&1 '&2' &3", ("Error! The file"), File.Name(sFilePath), ("cannot be
deleted!"))
  If Error Then Print Error.Class.Name;; Error.Code;; Error.Text;; Error.Where
  If Error Then Error "Error³!";; Error.Text
  If Error Then Error Error
  If Error Then Debug Error.Text
'-------------------------------------------------------------------------------------------------------
  If Error Then
     sMessage = "<b><font size='+1', color='DarkRed'>"
     sMessage &= ("Error")
     sMessage &= "</b></font><hr>"
     sMessage &= "The file '" & File.Name(sFilePath) & "' cannot be deleted.<br>"
     sMessage &= Error.Text & "!<br>"
     sMessage &= ("The error was raised in the source code line") & " " & Split(Error.Where, ".")[2] & "."
     Message.Error(sMessage)
  Endif
'-------------------------------------------------------------------------------------------------------
  Error Error
End
```

Es folgen die Ausgaben in der Konsole der IDE:

```
Error! The file `tmp.text` cannot be deleted!
Error! The file 'tmp.text' cannot be deleted!
FMain 45 File or directory does not exist FMain.btnKillFile_Click.10
Error³! File or directory does not exist
True
FMain.btnKillFile_Click.16: File or directory does not exist
True
```

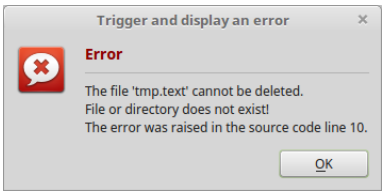

Abbildung 11.5.0.1.2: Fehler-Meldung

Kommentar

- Die Anweisung `Try Kill sFilePath` löst einen Fehler aus, wenn zum Beispiel die zu löschende Datei nicht existiert oder die Berechtigung zum Löschen der Datei fehlt.
- Im Beispiel zeigt der Dateipfad auf eine Datei, die nicht existiert, so dass garantiert ein Fehler ausgelöst wird. In dieser Syntax im o.a. Quelltext (man könnte sagen im "Auswertungskontext") verhält sich `Error` wie eine globale, nur lesbare Variable, die vom Interpreter verwaltet wird.
- Zuweisungen wie `Error = ...` sind aber nicht möglich und ergeben einen Syntaxfehler.
- Dieser Auswertungskontext umfasst Konstrukte wie `Variable = Error`, `Print Error` oder `If Error then ...`.
- Dem Auswertungskontext gegenüber steht der Anweisungskontext. Wenn `Error` syntaktisch an

einer Stelle steht, an der eine Anweisung erwartet wird, dann verhält es sich wie eine Anweisung. Die folgende Zeile dient nur der Demonstration:

```
If Error Then Error Error
```

- Das erste Error steht im "Auswertungskontext". Error steht für die o.a. boolsche Variable.
- Das zweite Error nach `Then` ist eine Anweisung – da es im Anweisungskontext steht.
- Das dritte Error steht für den Wert der boolschen Variablen Error.
- Kurzfassung: Wenn die Variable Error den Wert True hat, dann gibt die Anweisung Error den Wert Variablen Error aus – also erwartungsgemäß True.

11.5.0.1.5 Beispiel für den Einsatz der Methode Error.Raise(...)

Über die Eigenschaft Textbox.Text soll der Wert einer Basis für ein Zahlenwertsystem eingelesen werden, die im Programm verwendet werden soll. Für die Basis gilt: Basis ∈ ℤ (Datentyp Integer), Basis ≥ 2 und Basis ≤ 32. Hier sehen Sie den kompletten Quelltext für den Dialog:

```
' Gambas class file

Property Read Basis As Integer
Private $iBasis As Integer

Private Function Basis_Read() As Integer
  If FSetBasisDialog.ShowModal() = 1 Then
     Return $iBasis
  Else
     Return 0
  Endif
End

Public Sub Form_Open()
  FSetBasisDialog.Resizable = False
End

Public Sub txbBasis_Activate()
  SetBasis()
End

Public Sub btnOK_Click()
  SetBasis()
End

Private Function CheckInput(argInput As String) As Integer

  Dim iValue As Integer

  If Not argInput Then Error.Raise(("Die Eingabebox ist leer!"))
  If Not IsInteger(argInput) Then
     Error.Raise(("Der Text kann <b>nicht</b> in eine Integer-Zahl konvertiert werden!"))
  EndIf

  iValue = Val(argInput)

  If iValue < 2 Then Error.Raise(("Die Basis ist kleiner als 2!"))
  If iValue > 32 Then Error.Raise(("Die Basis ist größer als 32!"))

  Return iValue

End

Private Sub SetBasis()

  $iBasis = CheckInput(txbBasis.Text)
  FSetBasisDialog.Close(1)

  Catch
    Message.Error(Error.Text)
    Error.Clear()
    txbBasis.Clear()
    txbBasis.SetFocus()

End
```

Kommentar

- Die Validierung der Eingabe findet in der Funktion CheckInput(argument) statt.

- Auftretende Fehler werden vom Programmierer mit passend definierter Fehlermeldung mit der Methode *Error.Raise(error_description)* ausgelöst.
- Tritt bei der Zuweisung *$iBasis = CheckInput(txbBasis.Text)* ein Fehler auf, wird der Fehler mit CATCH sicher abgefangen und die dazu gehörende, benutzer-definierte Fehlermeldung ausgegeben, sonst wird der CATCH-Block übergangen.

11.5.0.1.6 Beispiel für den Einsatz der Methoden Error.Clear() und Error.Propagate()

Vorbemerkung: Im Quelltext von Gambas 3.12.2 finden Sie keine Zeile mit der Methode Error.Clear() und nur 3 mal wird die Methode Error.Propagate() verwendet. Das sagt ja wohl viel über den Stellenwert dieser beiden Methoden aus!

Wenn ein Fehler ausgelöst wurde, stoppt der Gambas-Interpreter die Ausführung des aktuellen Frames und sucht nach Fehlerbehandlern wie Try oder Catch. Wenn er diese nicht finden kann, geht der Interpreter den Aufrufstapel hoch und schaut dort nacheinander nach Fehlerbehandlern bis die globale Ebene erreicht ist. Dort sucht der Interpreter als letzten Ausweg die Methode `Static Public Sub Application_Error()` in der Start-Klasse. Wenn der Fehler nicht behoben werden konnte, gibt der Interpreter eine Fehlermeldung aus und beendet das Programm.

Um die automatische Abwicklung des Stapels zu beobachten, schauen Sie sich den folgenden Modul-Quelltext und die resultierenden Ausgaben an:

```
' Gambas module file

Public Sub Main()
    f()
  End

  Public Sub f()
    Try g()
    If Error Then
       Print "\nError-Backtrace:\n----------------"
       Print Error.Backtrace.Join("\n")
    Endif
  End

  Public Sub g()
    h(4)
    Print "Error from h(arg) will jump over this. Jumps to FINALLY"

    Finally
      Print "\nPassing through g()"
    ' Error.Clear()
      Error.Propagate() ' Apparently handles the error by triggering it again.
  End

  Public Sub h(x As Integer)
    Print "x = "; CStr(x); " | f(x) = "; 1 / x
    h(x - 1) ' Recursion - but without termination condition
    Print "Execution of h stops for x = 0 before we're here!"
  End
```

Ausgaben in der Konsole der IDE:

```
x = 4 | f(x) = 0,25
x = 3 | f(x) = 0,33333333333333
x = 2 | f(x) = 0,5
x = 1 | f(x) = 1

Passing through g()

Error-Backtrace:
----------------
Main.h.27
Main.h.28
Main.h.28
Main.h.28
Main.h.28
Main.g.16
Main.f.8
Main.Main.4
```

Man könnte sagen, dass die Anweisungen Try und Catch den Interpreter aus dem Ausnahmezustand nehmen. Das Abwickeln des Stapels wird gestoppt und die normale Ausführung fortgesetzt. Wenn Sie

aber mehrere "If Error Then"-Konstrukte im Quelltext haben, dann müssen Sie die Methode Error.Clear() einsetzen, nachdem Sie einen Fehler behandelt haben, um ihn nicht erneut zu behandeln! Kommentieren Sie zum Vergleich die Zeile mit dem Aufruf der Methode Error.Clear() aus und sehen Sie sich dann erneut die Ausgaben an!

11.5.0.1.7 Fehlernummern und Fehlermeldungen

Innerhalb einer Fehlerbehandlung können Sie auch auf die angezeigten Fehlernummern abzielen, die Sie über die Eigenschaft Error.Code der Klasse Error (gb) auslesen können. Eine vollständige Liste finden Sie unter http://gambaswiki.org/wiki/error. Nutzen Sie die Fehlernummern, um aussagekräftige Fehlermeldungen in deutscher Sprache zu erzeugen, denn hinter jeder Fehlernummer steht eine kurze Beschreibung des ausgelösten Fehlers in englischer Sprache, wie die folgenden Beispiele exemplarisch zeigen:

- (43) Access forbidden
- (44) File name is too long
- (45) File or directory does not exist
- (46) File is a directory
- (47) Read error
- (48) Write error

- Cannot find dynamic library (60)
- Cannot find symbol in dynamic library (61)
- Cannot load class (2)
- Cannot load component (27)
- Cannot open file (35)

Der folgende Quelltext-Ausschnitt setzt in der Select...Case-Kontrollstruktur im Catch-Block für ausgewählte Fehlernummern eine Fehlermeldung in deutscher Sprache. Die Fehlernummer wird aus der Eigenschaft *Error.Code* ausgelesen:

```
Public Sub btnSaveFile_Click()
 ' Speichert den Inhalt der TextArea1 in der Datei, zu dem der Pfad in TextBox1 führt
   Dim hFile As Stream
   Dim iCount As Integer
   Dim sErrorText As String

   hFile = Open TextBox1.Text For Write
   ...
     Write #hFile, bWrite, 1
   ...
 ' Den Stream auf jeden Fall schließen
   Finaly
     Close #hFile
   Catch
   Select Case Error.Code
     Case 43
       sErrorText = ("Sie besitzen nicht die Rechte,\num diese Datei zu speichern.")
     Case 44
       sErrorText = ("Der Dateiname ist zu lang.")
     Case 45
       sErrorText = ("Die Datei oder der Ordner existiert nicht.")
     Case 46
       sErrorText = ("Der Pfad führt zu einem Ordner.")
     Case 48
       sErrorText = ("Fehler beim Schreiben in die Datei.")
     Case Else
       sErrorText = ""
   End Select
 ' Ausgeben von Informationen über den ausgelösten Fehler
   Message.Error("Fehler-Code: " & Error.Code & "\n" & "bei " & Error.Where & "\n" & sErrorText)

End
```