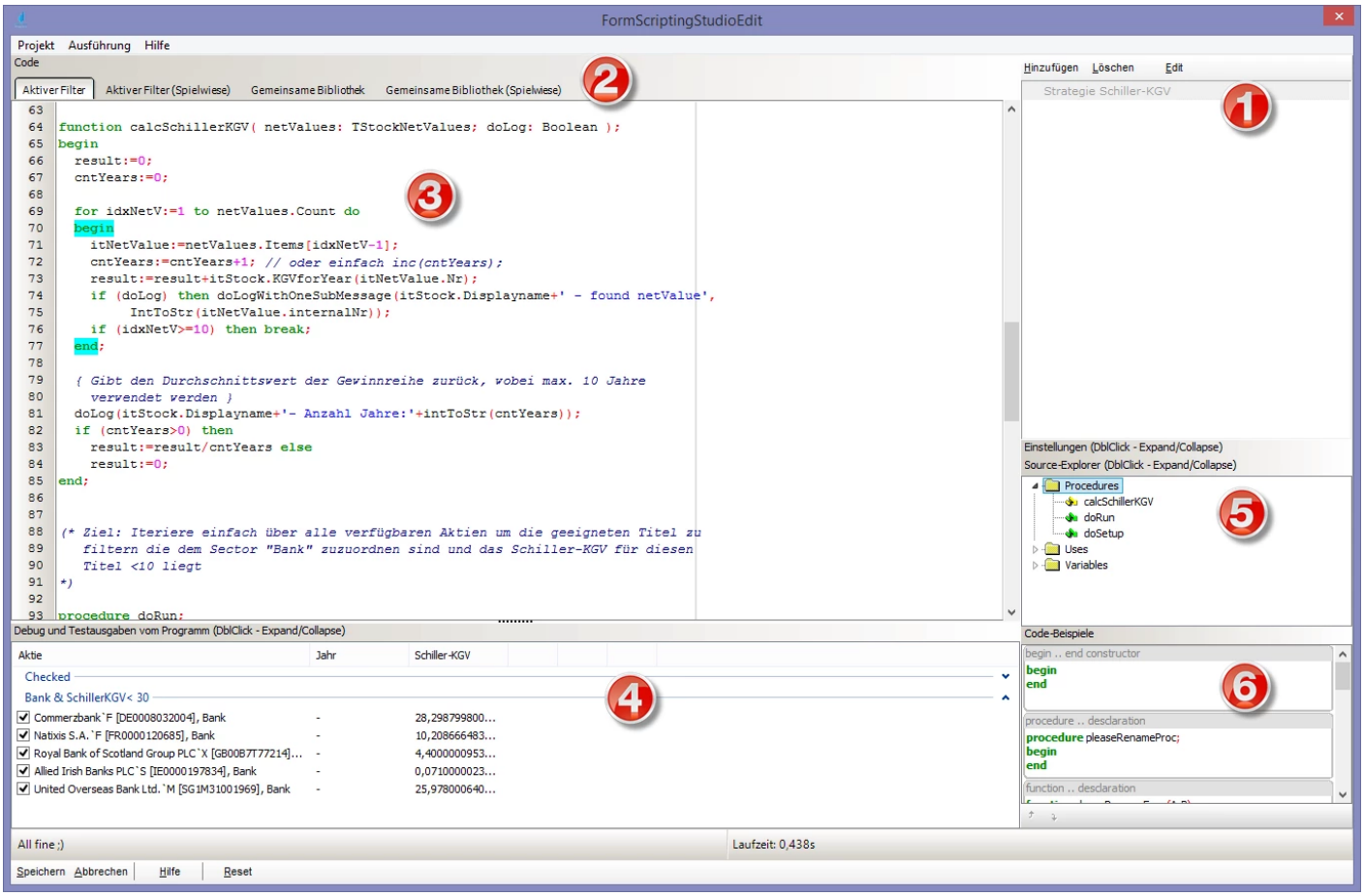
Grundlagen Handelsstrategie-Scripting

Die Scripting-Engine erweitert die bisherigen Filter-Module, um eine freie programmierbare Script-Engine. Es soll hiermit ein sehr hohen Freiheitsgrad in der Umsetzung eigener Filter gegeben werden und Routineaufgaben in der Titel-und Kurspflege automatisiert werden können.



Use-Cases

- Use-Case: Automatischer Export von Transaktionsdaten (Shareholder R/2 Börsensoftware)
- Use-Case: Automatische Watchliste auf Basis Indikator-Script (Shareholder R/2 Börsensoftware
- Use-Case: Automatische Stammdaten-Korrektur-Script (Shareholder R/2 Börsensoftware)

Handelsstrategie-Studio

Highlights

- Zugriff auf (alle) internen Datenobjekte
- Syntax-Highlighting
- Support einer produktiven und einer Entwicklungs-Version
- Support für Script-Bibliotheken, womit jedes Script von einem anderen aufgerufen werden kann. Dieser Mechanismus wird auch genutzt, um eine gemeinsame Bibliothek zu pflegen
- Hohe Ausführungsgeschwindigkeit (hier 0.4s über alle Aktien inkl. Log-Ausgaben)
- Source-Explorer

Überblick

Die Scripting-Engine führte freie geschriebene Scripte aus. Die Skripte sind in einem Pascal-Syntax, wobei spätere Alternativ-Sprachen denkbar sind (VB, Java, JS, Python). Der Einsatz der Scripting-Engine soll durchgängig durch SHAREholder vollzogen werden, d.h. eine Verwendung sowohl in Filter-Systemen, Druck-Templates als auch indirekt in den Handelssystemen ist vorgesehen. Der Startpunkt (v13.5) liegt hier bei einem Filtersystemen. Dabei ist ein wesentliches Feature der Zugriff auf alle relevanten internen Datenobjekte von SHAREholder. Es stehen damit beispielsweise folgende Funktionen zur Verfügung, die nicht nur lesenden Charakter haben:

- Automatische Anlegen einer Watchliste auf der Basis von Stammdaten-Filtern
- Automatische Bereinigung von Kommentaren in den vorhandenen Transaktionen, um hier nachträglich Ergänzungen vorzunehmen
- Automatische Bereinigung von Kursdaten
- Automatisches manuelles Erzeugen von Dateien (z.B. CSV-Export-Dateien) mit eigenen komplexeren Ausdrücken
- Anzeige von interessanten Titeln nach eigenen Regeln in Filterlisten

Folgender Syntax wird grundsätzlich unterstützt, womit komplette Programmstrukturen unterstützt werden:

```
begin .. end
procedure & function
if .. then .. else
for .. to .. do .. step
while .. do
repeat .. until
```

```
try .. except & try .. finally blocks
case statements
array constructors (x:=[ 1, 2, 3 ];)
^ , * , / , and , + , - , or , <> , >=, <= , = , > , < , div , mod , xor , shl , shroperators
access to object properties and methods ( ObjectName.SubObject.Property)
```

Damit sind in Summe komplexere Scripte in SHAREholder nutzbar für verschiedene Strategie-Filter-Umsetzungen aber auch für eigene Massenoperationen bis hin zu Im-Exportszenarien. Die in SHAREholder genutzt objektorientierte Programmierung kann dabei in Skripten werden, um auf Daten und Methoden hierarchisch zuzugreifen. Auch das Schreiben eigener Klassen ist möglich (siehe eigenen Abschnitt).

Editor

Der Editor setzt folgende Anforderungen um (Status v2.5):

- freie Codeeingabe
- Autovervollständigung
- Parameter-Darstellung d.h. innerhalb von Funktionen, Proceduren werden die erlaubten Parameter gezeigt
- Syntax-Highlighting auf die eingestellte Default-Sprache (hier Pascal/Delphi) und
- die Wiederverwendung von Code-Fragmenten durch Code-Bibliotheken
- Code-Folding d.h. das Ein/Ausklappen von Code-Fragmenten
- Snippet-Bibliothek, um sofort typische Fragmente nutzen zu können
- Automatischen Source-Explorer für aktuelle procedures und Variablen

Dokumentation SHAREholder interne Methoden und Klassen

- Eine Dokumentation der aktuell zur Verfügung stehenden Methoden die in den Scripten genutzt und referenziert werden können, finden Sie unter:

  https://www.shareholder24.com/docs/classes.html

Syntax der Scripte

Scripte enthalten a) procedure und function declarations und b) Haupt-Blöcke.

SCRIPT 1:

```
procedure DoSomething;
begin
  CallSomething;
end;

begin
  DoSomething;
end;
```

SCRIPT 2:

```
begin
  DoSomething;
end;
```

SCRIPT 3:

```
function MyFunction;
begin
  result:='Ok!';
end;
```

SCRIPT 4:

```
CallSomethingElse;
```

Statements sollten immer mit einem Semikolon ";" abgeschlossen werden.
begin..end Blöcke können jederzeit genutzt werden, um Statements zu gruppieren. Diese sind damit mit der Code-Folding-Funktion später auch ein-ausklappbar.

Identifier

Identifier Namen in Scripten (Variablen-Namen, function and procedure names, etc.) folgen folgenden einfachen Regeln:

- should begin with a character (a..z or A..Z), or '_', and can be followed by alphanumeric chars or '_' char.
- Cannot contain any other character os spaces.

Gültige Identifier:

```
VarName
_Some
V1A2
```

Einige ungültige Identifier:

```
2Var
My Name
Some-more
This,is,not,valid
```

Zuweisungen

Weisen Sie einen Wert oder ein Ausdrucksergebnis einer Variable oder Objekteigenschaft einfach mit ":=" zu.

```
MyVar := 2;
Button.Caption := 'This ' + 'is ok.';
```

Zeichenketten

Zeichenketten/Strings (Folge von Zeichen) werden in einfachen Anführungszeichen (') deklariert. Charakter. Doppelte Anführungszeichen (") werden nicht verwendet. Sie können auch " verwenden, als Zeichen innerhalb einer Zeichenkette. Es ist nicht notwendig, den '+'-Operator zu verwenden, um Zeichen-Codes zu einer Zeichenkette hinzuzufügen z.B. New-Lines mit Code: #13 und Code: #10. Einige Beispiele:

```
A := 'This is a text';
Str := 'Text '+'concat';
B := 'String with CR and LF char at the end'#13#10;
C := 'String with '#33#34' characters in the middle';
```

Kommentare

Kommentare können innerhalb des Skripts eingefügt werden.
Sie können

- //-Zeichen oder
- (* *) oder
- { } verwenden.

  Bei Blöcke mit // Zeichen wird der Kommentar am Ende der Zeile beendet.

```
//This is a comment before ShowMessage
ShowMessage('Ok');

(* This is another comment *)
ShowMessage('More ok!');

{ And this is a comment
with two lines }
ShowMessage('End of okays');
```

Variablen

Es ist nicht notwendig, Variablentypen in einem Skript zu deklarieren, obwohl Sie jede beliebige Art darin verwenden können. Sie können eine Variable nur mit der var-Direktive und ihrem Namen deklarieren:

```
procedure Msg;
var S;
begin
  S:='Hello world!';
  ShowMessage(S);
end;

var A;
begin
  A:=0;
  A:=A+1;
end;
```

```
var S: string;
begin
  S:='Hello World!';
  ShowMessage(S);
end;
```

Arrays

Auch wenn der Typ der Variablen nicht erforderlich ist und in den meisten Fällen ignoriert wird, gibt es einige spezielle Typen, die eine eigene Bedeutung haben. Sie können so Variablen als Array deklarieren und die Variable wird automatisch
als ein Variantenarray dieses Typs initialisiert:

```
var Arr: array[0..10] of string;
begin
  Arr[1] := 'first';
end;
```

Der Typ der Array-Elemente und der niedrige Index sind optional. Die nachfolgenden Beispiele sind alle gültig und resultieren in demselben Array-Typ:

```
var
  Arr1: array[0..10] of string;
  Arr2: array[10] of string;
  Arr3: array[0..10];
  Arr4: array[10];
```

Skript-Arrays beginnen immer bei 0, wenn nicht eine andere Start-Zahl festgelegt worden sind.

Es existiert eine Skriptunterstützung für Array-Konstruktoren und Unterstützung auch für Varianten-Arrays. Zum Konstruieren ein Array, verwenden Sie die Zeichen "[" und "]". Sie können eine Array-Verschachtelung mit mehreren Indizes konstruieren. Sie können dann über Indizes auf Arrays zugreifen. Wenn ein Array aus mehreren Indizes besteht, trennen Sie die Indizes mit ",". Wenn Variable ein Varianten-Array ist, unterstützt das Skript automatisch die Indizierung in diesem variabel. Eine Variable ist ein Variantenarray, wenn sie über ein Array zugewiesen wurde. Konstruktor, wenn es eine direkte Referenz auf eine Delphi-Variable ist, die eine Variante ist Array oder wenn es mit VarArrayCreate erstellt wurde

```
NewArray := [ 2,4,6,8 ];
Num:=NewArray[1]; //Num receives "4"
MultiArray := [ ['green','red','blue'] ,
['apple','orange','lemon'] ];
Str:=MultiArray[0,2]; //Str receives 'blue'
MultiArray[1,1]:='new orange';
```

Indexes

Strings, Arrays und Array-Eigenschaften können mit "[" und "]" Zeichen indiziert werden. Beispiel: Wenn Str eine Zeichenkettenvariable ist, gibt der Ausdruck Str[3] die dritte Zeichen in der durch Str bezeichneten Zeichenkette.

```
MyChar:=MyStr[2];
MyStr[1]:='A';
MyArray[1,2]:=1530;
Lines.Strings[2]:='Some text';
```

if-statements

Es gibt zwei Formen der if-Anweisung: if...then und die if...then...else. Wenn der if-Ausdruck wahr ist, wird die Anweisung (oder der Block) ausgeführt. Wenn es einen anderen Teil gibt und der Ausdruck falsch ist, wird die Anweisung (oder der Block) nach dem anderen ausführen.

```
if J <> 0 then Result := I/J;

if J = 0 then Exit else Result := I/J;

if J <> 0 then
begin
  Result := I/J;
  Count := Count + 1;
end else
  Done := True;
```

while-statements

Eine while-Anweisung wird verwendet, um eine Anweisung oder einen Block zu wiederholen, während eine Bedingung (Ausdruck) als wahr ausgewertet wird. Die Kontrollbedingung wird ausgewertet vor der Anweisungsfolge. Wenn also die Bedingung bei der ersten Iteration falsch ist, wird die Anweisungsfolge nie ausgeführt. Die while-Anweisung führt ihre konstituierende Aussage (oder Block) wiederholt aus, solange der Ausdruck in der Iteration gültig bleibt.

```
while Data[I] <> X do I := I + 1;
while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(InputFile) do
begin
  Readln(InputFile, Line);
  Process(Line);
end;
```

repeat statements

Die Syntax einer Wiederholungserklärung ist Wiederholungserklärung1; ...; Erklärung; bis Ausdruck, wobei expression einen booleschen Wert zurückgibt. Die Wiederholungsanweisung führt seine Sequenz von konstituierenden Aussagen kontinuierlich aus und testet Ausdruck nach jeder Iteration. Wenn der Ausdruck True zurückgibt, wird die Wiederholung Anweisung beendet. Die Sequenz wird immer mindestens einmal ausgeführt, weil Ausdruck wird erst nach der ersten Iteration ausgewertet. Beispiele:

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;

repeat
  Write('Enter a value (0..9): ');
```

```
  Readln(I);
until (I >= 0) and (I <= 9);
```

for statements

Scripter unterstützt Anweisungen mit folgender Syntax:

für Zähler :=initialValue bis finalValue
  do-Anweisung

Bei einer Anweisung, die den Zähler auf initialValue setzt, wird die Ausführung der Anweisung wiederholt (oder Block) und erhöhen den Wert des Zählers, bis der Zähler den Endwert erreicht hat.

```
for c:=1 to 10 do
  a:=a+c;

for i:=a to b do
begin
  j:=i^2;
  sum:=sum+j;
end;
```

case-statements

Der Scripter unterstützt Fallanweisungen mit folgender Syntax:

```
case selectorAusdruck von

  caseexpr1: Aussage1;
  ...
  caseexprn: Anweisung1; ..;

else

   elsestatement;
```

```
end
```

Wenn selectorAusruck mit dem Ergebnis eines der case-Ausdrücke übereinstimmt, wird die entsprechende Anweisung (oder der entsprechende Block) ausgeführt. Andernfalls wird das elsestatement ausgeführt. Ansonsten ist ein Teil der caseexprn-Anweisung optional. Anders als in Delphi muss die case-Anweisung im Skript nicht nur ordinale Werte verwenden. Sie können sowohl im Selektorausdruck als auch im Case-Ausdruck Ausdrücke beliebigen Typs verwenden. Beispiel:

```
case uppercase(Fruit) of
   'lime': ShowMessage('green');
   'orange': ShowMessage('orange');
   'apple': ShowMessage('red');
else
   ShowMessage('black');
end;
```

function and procedure declaration

Die Deklaration von Funktionen und Prozeduren ist ähnlich wie bei Object Pascal in Delphi, mit dem Unterschied, dass man keine Variablentypen angibt. Genau wie bei OP verwenden Sie implizit deklarierte Ergebnisvariablen, um Funktionswerte zurückzugeben. Es können auch Parameter per Referenz verwendet werden, mit der erwähnten Einschränkung: es ist nicht notwendig, Variablentypen anzugeben. Für eine einfacher Lesbarkeit kann dies jedoch genutzt werden ergänzend zu einer Namensgebung, die den Ergebnis-Typ mit in den Funktionsnamen aufnimmt z.B. TodayAsString ,TodayAsDate ...

```
procedure HelloWord;
begin
   ShowMessage('Hello world!');
end;

procedure UpcaseMessage(Msg);
begin
   ShowMessage(Uppercase(Msg));
end;

function TodayAsString;
begin
```

```
   result:=DateToStr(Date);
end;

function Max(A,B);
begin
   if A>B then
      result:=A
   else
      result:=B;
end;

procedure SwapValues(var A, B);
Var Temp;
begin
   Temp:=A;
   A:=B;
   B:=Temp;
end;
```

Script-based libraries

Skript-basierte Bibliothek ist das Konzept, bei dem ein Skript andere Skripte "benutzen" kann (zum Aufruf von Prozeduren, zum Setzen globaler Variablen usw.).

Nehmen Sie zum Beispiel die folgenden Skripte:

```
//Script 1
uses Script2;

begin
  Script2GlobalVar := 'Hello world!';
  ShowScript2Var;
end;



//Script2
var
```

```
  Script2GlobalVar: string;

procedure ShowScript2Var;
begin
  ShowMessage(Script2GlobalVar);
end;
```

Wenn Sie das erste Skript ausführen, "benutzt" es Script2, und dann ist es in der Lage, globale Variablen zu lesen/schreiben und Prozeduren aus Script2 aufzurufen.

Das einzige Problem hier ist, dass Skript 1 "wissen" muss, wo es Skript2 finden kann.

Wenn der Compiler z.B. einen Bezeichner in der uses-Klausel erreicht, muss er wissen, wo es zu finden ist:

```
uses Classes, Forms, Script2;
```

Hier wird nun folgender Ablauf abgebildet, um die Bibliothek auf verschiedene Weise zu "laden":

1. Er versucht, eine registrierte Delphi-basierte Bibliothek mit diesem Namen zu finden.

Mit anderen Worten, jede Bibliothek, die bei RegisterScripterLibrary registriert wurde. Dies gilt sowohl für die importierte VCL, die mit Scripter Studo bereitgestellt wird, als auch für die vom Import-Tool importierten Klassen. Dies ist der Fall für Klassen, Formulare und andere Einheiten.

2. Versucht, ein Skript in der Scripts-Sammlung zu finden, bei dem UnitName mit dem Bibliotheksnamen übereinstimmt.

Jedes TatScript-Objekt in der Scripter.Scripts-Sammlung verfügt über eine UnitName-Eigenschaft. Sie können diese Eigenschaft manuell einstellen, so dass das Skript-Objekt in diesen Situationen wie eine Bibliothek behandelt wird. Im obigen Beispiel könnten Sie ein Skript-Objekt hinzufügen, seine SourceCode-Eigenschaft auf den Skript-2-Code setzen und dann UnitName auf 'Skript2' setzen. Auf diese Weise könnte das Skript1 das Skript2 als Bibliothek finden und seine Variablen und Funktionen verwenden.

3. Versucht, eine Datei zu finden, deren Name mit dem Bibliotheksnamen übereinstimmt (wenn LibOptions.UseScriptFiles auf 'true' gesetzt ist)

Wenn LibOptions.UseScriptFiles auf true gesetzt ist, versucht der Skripter, die Bibliothek in Dateien zu finden. Wenn das Skript z. B. "uses Script2;" hat, sucht es nach Dateien mit dem Namen "Script2.psc".

Declaring classes in script (script-based classes)

Es ist jetzt möglich, Klassen in einem Skript zu deklarieren. Mit dieser Funktion können Sie eine Klasse deklarieren, um sie auf ähnliche Weise wie in Delphi zu verwenden: Sie erstellen eine Instanz der Klasse und verwenden sie wieder.

Jede Klasse muss in einem separaten Skript deklariert werden, d.h. Sie müssen für jede zu deklarierende Klasse ein Skript haben.

Sie machen das Skript zu einem "Klassenskript", indem Sie die $CLASS-Direktive am Anfang des Skripts, gefolgt vom Klassennamen, hinzufügen:

```
//Turn this script into a class script for TSomeClass
{$CLASS TSomeClass}
```

Methoden und Eigenschaften

Jede in einem Klassenskript deklarierte globale Variable wird tatsächlich zu einer Eigenschaft der Klasse. Jede Prozedur/Funktion in einem Skript wird zu einer Klassenmethode.

Die Hauptroutine des Skripts wird immer dann ausgeführt, wenn eine neue Instanz der Klasse erzeugt wird, so dass sie als Klasseninitialisierer verwendet werden kann und Sie einige Eigenschaften auf Standardwerte setzen und eine ordnungsgemäße Klasseninitialisierung durchführen können.

```
//My class script
{$CLASS TMyClass}
uses Dialogs;

var
  MyProperty: string;

procedure SomeMethod;
begin
  ShowMessage('Hello, world!');
end;
```

```
// class initializer
begin
  MyProperty := 'Default Value';
end;
```

Nutzung der Klassen

Sie können die Klasse aus anderen Skripten heraus verwenden, indem Sie einfach eine neue Instanz der genannten Klasse erzeugen:Sie können die Klasse aus anderen Skripten heraus verwenden, indem Sie einfach eine neue Instanz der genannten Klasse erzeugen:

```
uses MyClassScript;
var
  MyClass: TMyClass;
begin
  MyClass := TMyClass.Create;
  MyClass.MyProperty := 'test';
  MyClass.SomeMethod;
end;
```

Einzelheiten der Implementierung

Die im Skript deklarierten Klassen sind "Pseudo"-Klassen. Das bedeutet, dass keine neuen Delphi-Klassen erstellt werden. Obwohl Sie z.B. in dem obigen Beispiel TMyClass.Create aufrufen, der Name "TMyClass" nur für das Skriptsystem bedeutet, gibt es keine Delphi-Klasse mit dem Namen TMyClass. Alle Objekte, die als skriptbasierte Klassen erstellt werden, sind eigentlich Instanzen der Klasse TScriptBaseObject. Sie können dieses Verhalten ändern, um Instanzen einer anderen Klasse zu erzeugen, aber diese neue Klasse muss von der Klasse TScriptBaseObject erben. Sie definieren die Basisklasse für alle "Pseudo"-Klassenobjekte in der Scripter-Eigenschaft ScriptBaseObjectClass.

Speicherverwaltung

Obwohl Sie in Skripten die Methode Free aufrufen können, um Speicher freizugeben, der mit Instanzen von skriptbasierten Klassen verbunden ist, müssen Sie das nicht tun.

Alle in Skripten erstellten Objekte, die auf Skript-Klassen basieren, werden schließlich von der Skripter-Komponente zerstört.

Einschränkungen

Da Scripter keine neuen echten Delphi-Klassen erstellt, gibt es einige Einschränkungen, was Sie damit machen können. Die wichtigste ist, dass Vererbung nicht unterstützt wird. Da alle Klassen in Skripter tatsächlich dieselbe Delphi-Klasse sind, können Sie keine Klassen erstellen, die von einer anderen Delphi-Klasse erben, außer der in der Klasse TScriptBaseObject deklarierten.

Revision #4
Created 2022-06-19 07:34:59 UTC by Jens Werschmoeller
Updated 2024-10-03 07:06:36 UTC by Jens Werschmoeller