



# Быстрые отчеты

# Руководство разработчика FastScript

Версия 2024.2

© 2008-2024 ООО Быстрые отчеты

# Что такое FastScript

FastScript - библиотека для выполнения скриптов. Она будет полезна разработчикам, желающим добавить возможности исполнения скриптовых программ в свои проекты. FastScript написан полностью на 100% Object Pascal и может быть установлен в Delphi и C++Builder.

Уникальные возможности FastScript - возможность одновременного использования нескольких языков (в настоящее время - PascalScript, C++Script, BasicScript и JScript), вы можете писать скрипты, используя ваш любимый язык программирования. FastScript не использует Microsoft Scripting Host, а потому может использоваться как в Windows, так и в Linux.

FastScript объединяет в себе кросс-платформенность, быстрое выполнение кода, компактность, богатый выбор возможностей и великолепную масштабируемость. Сделайте ваши приложения максимально гибкими и мощными с FastScript!

# Быстрый старт

Вот пример кода, который демонстрирует простейший способ использования FastScript. Для корректной работы примера положите на форму компоненты `fsScript1: TfsScript` и `fsPascal1: TfsPascal`.

```
uses FS_iInterpreter;

procedure TForm1.Button1Click(Sender: TObject);
begin
  fsScript1.Clear; // необходимо очищать компонент, если он используется для запуска
  fsScript1.Lines.Text := 'begin ShowMessage(''Hello!'') end.';
  fsScript1.Parent := fsGlobalUnit; // use standard classes and methods
  fsScript1.SyntaxType := 'PascalScript';
  if fsScript1.Compile then
    fsScript1.Execute else
    ShowMessage(fsScript1.ErrorMsg);
end;
```

Как видите, здесь нет ничего сложного. Мы заполняем свойство `fsScript1.Lines` текстом скрипта. Затем мы указываем, что наш скрипт будет использовать стандартные типы и функции, определенные в глобальном модуле `fsGlobalUnit`. После этого мы компилируем скрипт, используя язык `PascalScript`. Если компиляция успешна, метод `Compile` возвратит `True` и мы можем выполнить (`Execute`) скрипт. Иначе будет выведено сообщение об ошибке.

# Реализованные и нереализованные особенности

## Реализованные особенности

- мультязычная архитектура, позволяющая использовать множество языков (в настоящее время - PascalScript и C++Script). Можете добавлять любые другие процедурно-ориентированные языки (их описание хранится в XML-формате)
- стандартный языковой набор: переменные, константы, процедуры, функции (с возможностью вложенности) с переменными/постоянными/умалчиваемыми параметрами, все стандартные операторы и объявления (включая case, try/finally/except, with), типы (целый, дробный, логический, символьный, строковый, многомерные массивы, множество, variant), классы (с методами, событиями, свойствами, индексами и свойствами по умолчанию).
- проверка совместимости типов.
- доступ к любому объекту вашего приложения. Стандартные библиотеки для доступа к базовым классам, контролам, формам и БД. Легко расширяемая архитектура библиотеки.
- Компактность - 90-150Кб в зависимости от используемых модулей.

## Нереализованные особенности

- Отсутствуют объявления типов (records, classes) в скрипте; нет записей (records), указателей (pointers), множеств (sets) (однако возможно использование оператора 'IN' - "a in ['a'..'c','d']"), нет типа shortstrings, нет безусловного перехода (GOTO).
- C++Script: нет восьмеричных констант; нет 'break' в операторе SWITCH (SWITCH работает подобно Pascal CASE); операторы '++' и '--' возможны только после переменных, т.е. '++i' не будет работать; операторы '--', '++' и '=' ничего не возвращают, т.е. 'if(++)' не будет работать; все идентификаторы не чувствительны к регистру; Константа NULL это Null из Pascal- используйте nil вместо NULL.
- JScript и BasicScript: см. синтаксические диаграммы.

# Описание языка

## Синтаксис PascalScript:

```
Program -> [PROGRAM Ident ';' ]
          [UsesClause]
          Block '.'

UsesClause -> USES (String/,)... ';'

Block -> [DeclSection]...
        CompoundStmt

DeclSection -> ConstSection
             -> VarSection
             -> ProcedureDeclSection

ConstSection -> CONST (ConstantDecl)...

ConstantDecl -> Ident '=' Expression ';'

VarSection -> VAR (VarList ';' )...

VarList -> Ident/','... ':' TypeIdent [InitValue]

TypeIdent -> Ident
           -> Array

Array -> ARRAY '[' ArrayDim/','... ']' OF Ident

ArrayDim -> Expression..Expression
          -> Expression

InitValue -> '=' Expression

Expression -> SimpleExpression [RelOp SimpleExpression]...

SimpleExpression -> ['-'] Term [AddOp Term]...

Term -> Factor [MulOp Factor]...

Factor -> Designator
        -> UnsignedNumber
        -> String
        -> '(' Expression ')'
        -> NOT Factor
        -> '[' SetConstructor ']'

SetConstructor -> SetNode/','...

SetNode -> Expression ['..' Expression]

RelOp -> '>'
        -> '<'
        -> '<='
        -> '>='
        -> '<>'
        -> '='
        -> IN
        -> IS

AddOp -> '+'
        -> '-'
```

```

-> OR
-> XOR

MulOp -> '*'
-> '/'
-> DIV
-> MOD
-> AND
-> SHL
-> SHR

Designator -> ['@'] Ident ['. ' Ident | '[' ExprList ']' | '(' ExprList ')']...

ExprList -> Expression/','...

Statement -> [SimpleStatement | StructStmt]

StmtList -> Statement/';'...

SimpleStatement -> Designator
                  -> Designator ':=' Expression
                  -> BREAK | CONTINUE | EXIT

StructStmt -> CompoundStmt
              -> ConditionalStmt
              -> LoopStmt
              -> TryStmt
              -> WithStmt

CompoundStmt -> BEGIN StmtList END

ConditionalStmt -> IfStmt
                  -> CaseStmt

IfStmt -> IF Expression THEN Statement [ELSE Statement]

CaseStmt -> CASE Expression OF CaseSelector/';'... [ELSE Statement] [';'] END

CaseSelector -> SetConstructor ':' Statement

LoopStmt -> RepeatStmt
            -> WhileStmt
            -> ForStmt

RepeatStmt -> REPEAT StmtList UNTIL Expression

WhileStmt -> WHILE Expression DO Statement

ForStmt -> FOR Ident ':=' Expression ToDownto Expression DO Statement

ToDownto -> (TO | DOWNTO)

TryStmt -> TRY StmtList (FINALLY | EXCEPT) StmtList END

WithStmt -> WITH (Designator/,..) DO Statement

ProcedureDeclSection -> ProcedureDecl
                       -> FunctionDecl

ProcedureDecl -> ProcedureHeading ';'
                Block ';'

ProcedureHeading -> PROCEDURE Ident [FormalParameters]

FunctionDecl -> FunctionHeading ';'
               Block ';'

FunctionHeading -> FUNCTION Ident [FormalParameters] ':' Ident

```

FormalParameters -> '(' FormalParam/','... ')'

FormalParm -> [VAR | CONST] VarList

## Синтаксис C++Script:

```
Program -> [UsesClause]
         [DeclSection]...
         CompoundStmt

UsesClause -> '#' INCLUDE (String/,)...

DeclSection -> ConstSection
             -> ProcedureDeclSection
             -> VarStmt ';'

ConstSection -> '#' DEFINE ConstantDecl

ConstantDecl -> Ident Expression

VarStmt -> Ident Ident [Array] [InitValue] /','...

ArrayDef -> '[' ArrayDim/','... ']'

ArrayDim -> Expression

InitValue -> '=' Expression

Expression -> SimpleExpression [RelOp SimpleExpression]...

SimpleExpression -> ['-'] Term [AddOp Term]...

Term -> Factor [MulOp Factor]...

Factor -> Designator
        -> UnsignedNumber
        -> String
        -> '(' Expression ')'
        -> '!' Factor
        -> '[' SetConstructor ']'
        -> NewOperator

SetConstructor -> SetNode/','...

SetNode -> Expression ['..' Expression]

NewOperator -> NEW Designator

RelOp -> '>'
        -> '<'
        -> '<='
        -> '>='
        -> '!='
        -> '=='
        -> IN
        -> IS

AddOp -> '+'
        -> '-'
        -> '||'
        -> '^'

MulOp -> '*'
        -> '/'
        -> '%'
        -> '&&'
```

```

-> '<<'
-> '>>'

Designator -> ['&'] Ident ['. ' Ident | '[' ExprList ']' | '(' ExprList ')']...

ExprList -> Expression / ',' ...

Statement -> [SimpleStatement ';' | StructStmt | EmptyStmt]

EmptyStmt -> ';'

StmtList -> (Statement...)

SimpleStatement -> DeleteStmt
                  -> AssignStmt
                  -> VarStmt
                  -> CallStmt
                  -> ReturnStmt
                  -> (BREAK | CONTINUE | EXIT)

DeleteStmt -> DELETE Designator

AssignStmt -> Designator ['+' | '-' | '*' | '/'] '=' Expression

CallStmt -> Designator ['++' | '--']

ReturnStmt -> RETURN [Expression]

StructStmt -> CompoundStmt
            -> ConditionalStmt
            -> LoopStmt
            -> TryStmt

CompoundStmt -> '{' [StmtList] '}'

ConditionalStmt -> IfStmt
                -> CaseStmt

IfStmt -> IF '(' Expression ')' Statement [ELSE Statement]

CaseStmt -> SWITCH '(' Expression ')' '{' (CaseSelector)... [DEFAULT ':' Statement] '}'

CaseSelector -> CASE SetConstructor ':' Statement

LoopStmt -> RepeatStmt
          -> WhileStmt
          -> ForStmt

RepeatStmt -> DO Statement [';'] WHILE '(' Expression ')' ';'

WhileStmt -> WHILE '(' Expression ')' Statement

ForStmt -> FOR '(' ForStmtItem ';' Expression ';' ForStmtItem ')' Statement

ForStmtItem -> AssignStmt
             -> VarStmt
             -> CallStmt
             -> Empty

TryStmt -> TRY CompoundStmt (FINALLY | EXCEPT) CompoundStmt

FunctionDecl -> FunctionHeading CompoundStmt

FunctionHeading -> Ident Ident [FormalParameters]

FormalParameters -> '(' [FormalParam / ';' ...] ')'

FormalParam -> TypeIdent (['&'] Ident [InitValue] / ',')...

```



## Синтаксис JScript:

```
Program -> Statements

Statements -> Statement...

Block -> '{' Statements '}'

ImportStmt -> IMPORT (String/,)...

VarStmt -> VAR (VarDecl/',')...

VarDecl -> Ident [Array] [InitValue]

Array -> '[' (ArrayDim/',')... ']'

ArrayDim -> Expression

InitValue -> '=' Expression

Expression -> SimpleExpression [RelOp SimpleExpression]...

SimpleExpression -> ['-'] Term [AddOp Term]...

Term -> Factor [MulOp Factor]...

Factor -> Designator
        -> UnsignedNumber
        -> String
        -> '(' Expression ')'
        -> '!' Factor
        -> '[' SetConstructor ']'
        -> NewOperator
        -> '<' FRString '>'

SetConstructor -> SetNode/','...

SetNode -> Expression ['..' Expression]

NewOperator -> NEW Designator

RelOp -> '>'
        -> '<'
        -> '<='
        -> '>='
        -> '!='
        -> '=='
        -> IN
        -> IS

AddOp -> '+'
        -> '-'
        -> '||'
        -> '^'

MulOp -> '*'
        -> '/'
        -> '%'
        -> '&&'
        -> '<<'
        -> '>>'

Designator -> ['&'] Ident ['.' Ident | '[' ExprList ']' | '(' [ExprList] ')']...

ExprList -> Expression/','...
```



```

InitValue -> '=' Expression

Expression -> SimpleExpression [RelOp SimpleExpression]...

SimpleExpression -> ['-'] Term [AddOp Term]...

Term -> Factor [MulOp Factor]...

Factor -> Designator
        -> UnsignedNumber
        -> String
        -> '(' Expression ')'
        -> NOT Factor
        -> NewOperator
        -> '<' FRString '>'

SetConstructor -> SetNode/','...

SetNode -> Expression ['..' Expression]

NewOperator -> NEW Designator

RelOp -> '>'
        -> '<'
        -> '<='
        -> '>='
        -> '<>'
        -> '='
        -> IN
        -> IS

AddOp -> '+'
        -> '-'
        -> '&'
        -> OR
        -> XOR

MulOp -> '*'
        -> '/'
        -> '\'
        -> MOD
        -> AND

Designator -> [ADDRESSOF] Ident ['.' Ident | '[' ExprList ']' | '(' [ExprList] ')']...

ExprList -> Expression/','...

Statement -> BreakStmt
            -> CaseStmt
            -> ContinueStmt
            -> DeleteStmt
            -> DimStmt
            -> DoStmt
            -> ExitStmt
            -> ForStmt
            -> FuncStmt
            -> IfStmt
            -> ImportStmt
            -> ProcStmt
            -> ReturnStmt
            -> SetStmt
            -> TryStmt
            -> WhileStmt
            -> WithStmt
            -> AssignStmt
            -> CallStmt

BreakStmt -> BREAK

```

```

ContinueStmt -> CONTINUE

ExitStmt -> EXIT

DeleteStmt -> DELETE Designator

SetStmt -> SET AssignStmt

AssignStmt -> Designator ['+'|'-'|'*'|'/'] '=' Expression

CallStmt -> Designator ['+++'|'--'-']

ReturnStmt -> RETURN [Expression]

IfStmt -> IF Expression THEN ThenStmt

ThenStmt -> EOL [Statements] [ElseIfStmt | ElseStmt] END IF
-> StatementList

ElseIfStmt -> ELSEIF Expression THEN
(EOL [Statements] [ElseIfStmt | ElseStmt] | Statement)

ElseStmt -> ELSE (EOL [Statements] | Statement)

CaseStmt -> SELECT CASE Expression EOL
(CaseSelector...) [CASE ELSE ':' Statements] END SELECT

CaseSelector -> CASE SetConstructor ':' Statements

DoStmt -> DO [Statements] LOOP (UNTIL | WHILE) Expression

WhileStmt -> WHILE Expression [Statements] WEND

ForStmt -> FOR Ident '=' Expression TO Expression [STEP Expression] EOL
[Statements] NEXT

TryStmt -> TRY Statements (FINALLY | CATCH) [Statements] END TRY

WithStmt -> WITH Designator EOL Statements END WITH

ProcStmt -> SUB Ident [FormalParameters] EOL [Statements] END SUB

FuncStmt -> FUNCTION Ident [FormalParameters] [AsClause] EOL
[Statements] END FUNCTION

FormalParameters -> '(' (FormalParam/','...) ')'

FormalParm -> [BYREF | BYVAL] VarList

```

# Структура скрипта

Структура **PascalScript** почти такая же, как и у Object Pascal:

```
#language PascalScript // опционально
program MyProgram;     // опционально

uses 'unit1.pas', 'unit2.pas';
// раздел uses должен быть перед любыми другими разделами
// v1.2 changes: Внимание! Теперь подключаемые модули НЕ вставляются в текст главного модуля.
// Таким образом, они могут иметь секции 'program', 'uses' и должны иметь секцию 'main procedure'.

var                    // раздел var
  i, j: Integer;

const                 // раздел const
  pi = 3.14159;

procedure p1;        // процедуры и функции
var
  i: Integer;

  procedure p2;      // вложенная процедура
  begin
  end;

begin
end;

begin                // главный исполняемый модуль.
end.
```

Структура **C++Script**:

```
#language C++Script // опционально
#include "unit1.cpp", "unit2.cpp"
// раздел include - должен быть перед любым другим разделом

int i, j;            // раздел переменных - может быть в любом месте

#define pi = 3.14159 // раздел констант

void p1()            // функции
{                    // вложенных процедур нет
}

{                    // главная исполняемая функция.
}
```

Структура **JScript**:

```

#language JScript // опционально
import "unit1.js", "unit2.js"
// раздел import - должен быть перед любым другим разделом

var i, j = 0;           // раздел переменных - может быть в любом месте

function p1()          // функции
{                      //
}

                        // главная исполняемая функция.
p1();
for (i = 0; i < 10; i++) j++;

```

### Структура **BasicScript**:

```

#language BasicScript 'опционально
imports "unit1.vb", "unit2.vb"
' раздел imports - должен быть перед любым другим разделом

dim i, j = 0           ' раздел переменных - может быть в любом месте

function f1()          ' функции
end function

sub p1()
end sub

                        ' главная исполняемая функция.
for i = 0 to 10
  p1()
next

```

# Типы данных

FastScript работает с типом Variant и основан на нём. Тем не менее, вы можете использовать следующие предопределённые типы в ваших скриптах:

Byte		целочисленные
Word		
Integer		
Longint		
Cardinal		
TColor		
Boolean		логический
Real		расширенный (с плавающей запятой)
Single		
Double		
Extended		
TDate		
TTime		
TDateTime		
Char		символьный
String		строковый
Variant		Variant (вариантный тип)
Pointer		
Array		массив

Соответствие некоторых типов **C++Script** стандартным типам:

```
int, long = Integer
void = Integer
bool = Boolean
float = Extended
```

**JScript** не имеет описаний типов - все типы являются Variant. BasicScript может использовать описание типов (напр. `dim i as Integer`), а может опускать тип или даже объявление переменной. В этом случае она считается типом Variant.

Не все из этих типов могут быть неявно приведены один к другому. Как и в Object Pascal, вы не можете привести Extended или String к Integer. Только один тип - Variant - может быть присвоен любому типу и получить значение от любого типа.

Помимо встроенных типов, вы можете использовать перечислимые типы, объявленные в вашем приложении или в дополнительных модулях (к примеру, добавив компонент TfsGraphicsRTTI, вы сможете использовать TPenMode, TFontStyles и другие типы).

# Классы

Вы не можете объявить класс в скрипте, но вы можете использовать внешние классы, объявленные в вашем приложении или в дополнительных модулях. Вот демонстрационный пример DEMOS\Main:

```
var
  f: TForm;
  b: TButton;

procedure ButtonClick(Sender: TButton);
begin
  ShowMessage(Sender.Name);
  f.ModalResult := mrOk;
end;

// нет никакой необходимости использовать все параметры в обработчиках событий,
// потому что здесь не производится никакого контроля соответствия типов

procedure ButtonMouseMove(Sender: TButton);
begin
  b.Caption := 'moved over';
end;

begin
  f := TForm.Create(nil);
  f.Caption := 'Test it!';
  f.BorderStyle := bsDialog;
  f.Position := poScreenCenter;

  b := TButton.Create(f);
  b.Name := 'Button1';
  b.Parent := f;
  b.SetBounds(10, 10, 75, 25);
  b.Caption := 'Test';

  b.OnClick := @ButtonClick; { same as b.OnClick := 'ButtonClick' }
  b.OnMouseMove := @ButtonMouseMove;

  f.ShowModal;
  f.Free;
end.
```

Как видите, нет никакой разницы между PascalScript и кодом Delphi. Вы можете получить доступ к любому свойству (простому, индексному или умалчиваемому) или методу. По умолчанию все published свойства объектов доступны из скрипта. Public свойства и методы нуждаются в implementation коде - поэтому вы можете получить к ним частичный доступ (к примеру, вы не можете получить доступ к методу TForm.Print или свойству TForm.Canvas, поскольку они не реализованы).

Вы можете добавлять ваши собственные классы - подробнее смотрите главу "создание скриптов".



# Функции

В скрипте можно использовать богатейший набор стандартных функций. To get an access to these functions, pass the `fsGlobalUnit` reference to the `TfsScript.Parent` property.

```
function IntToStr(i: Integer): String
function FloatToStr(e: Extended): String
function DateToStr(e: Extended): String
function TimeToStr(e: Extended): String
function DateTimeToStr(e: Extended): String
function VarToStr(v: Variant): String

function StrToInt(s: String): Integer
function StrToFloat(s: String): Extended
function StrToDate(s: String): Extended
function StrToTime(s: String): Extended
function StrToDateTime(s: String): Extended

function Format(Fmt: String; Args: array): String
function FormatFloat(Fmt: String; Value: Extended): String
function FormatDateTime(Fmt: String; DateTime: TDateTime): String
function FormatMaskText(EditMask: string; Value: string): string

function EncodeDate(Year, Month, Day: Word): TDateTime
procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word)
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word)
function Date: TDateTime
function Time: TDateTime
function Now: TDateTime
function DayOfWeek(aDate: DateTime): Integer
function IsLeapYear(Year: Word): Boolean
function DaysInMonth(nYear, nMonth: Integer): Integer

function Length(s: String): Integer
function Copy(s: String; from, count: Integer): String
function Pos(substr, s: String): Integer
procedure Delete(var s: String; from, count: Integer): String
procedure Insert(s: String; var s2: String; pos: Integer): String
function Uppercase(s: String): String
function Lowercase(s: String): String
function Trim(s: String): String
function NameCase(s: String): String
function CompareText(s, s1: String): Integer
function Chr(i: Integer): Char
function Ord(ch: Char): Integer
procedure SetLength(var S: String; L: Integer)

function Round(e: Extended): Integer
function Trunc(e: Extended): Integer
function Int(e: Extended): Integer
function Frac(X: Extended): Extended
function Sqrt(e: Extended): Extended
function Abs(e: Extended): Extended
function Sin(e: Extended): Extended
function Cos(e: Extended): Extended
function ArcTan(X: Extended): Extended
function Tan(X: Extended): Extended
function Exp(X: Extended): Extended
function Ln(X: Extended): Extended
function Pi: Extended

procedure Inc(var i: Integer; incr: Integer = 1)
```

```
procedure Dec(var i: Integer; decr: Integer = 1)
procedure RaiseException(Param: String)
procedure ShowMessage(Msg: Variant)
procedure Randomize
function Random: Extended
function ValidInt(cInt: String): Boolean
function ValidFloat(cFlt: String): Boolean
function ValidDate(cDate: String): Boolean
function CreateOleObject(ClassName: String): Variant
function VarArrayCreate(Bounds: Array; Typ: Integer): Variant
```

Как видите, некоторые функции и процедуры содержат параметры по умолчанию. Вы можете вызывать их так же, как и в Delphi:

```
Inc(a);
Inc(b, 2);
```

Вы можете подключить свои собственные процедуры и функции к скрипту - подробнее смотрите главу "создание скриптов".

# События

Вы можете использовать обработчики событий в скрипте. В отличие от обработчиков событий в Delphi, обработчики событий в скрипте не являются методами объекта. Следующий пример показывает, как подключить обработчик события к событию TButton.OnClick:

```
var
  b: TButton;
  Form1: TForm;

procedure ButtonClick(Sender: TButton);
begin
  ShowMessage(Sender.Name);
end;

begin
  b := TButton.Create(Form1);
  b.Parent := Form1;
  b.OnClick := @ButtonClick; { тоже что и b.OnClick := 'ButtonClick' }
  b.OnClick := nil; // сброс обработчика
end.
```

Вот некоторые predefined события, доступные в модуле FS\_iEvents:

```
TfsNotifyEvent
TfsMouseEvent
TfsMouseMoveEvent
TfsKeyEvent
TfsKeyPressEvent
TfsCloseEvent
TfsCloseQueryEvent
TfsCanResizeEvent
```

Смотрите главы "Компонент TfsFormsRTTI ", "Компонент TfsExtCtrlsRTTI " и "Компонент TfsDBCtrlsRTTI " для получения списка доступных событий.

# Перечисления и множества

**FastScript** поддерживает перечисления. Вы можете написать в скрипте:

```
Form1.BorderStyle := bsDialog;
```

Множества не поддерживаются. Тем не менее, вы можете оперировать элементами множества:

```
Font.Style := fsBold;           // Font.Style := [fsBold] in Delphi  
Font.Style := fsBold + fsItalic; // Font.Style := [fsBold, fsItalic]  
Font.Style := 0;                // Font.Style := []
```

# Массивы

**FastScript** поддерживает все типы массивов: статические (одномерные, многомерные), динамические, варианты. Вот пример скрипта, использующего три массива целых чисел, объявленных разным способом:

```
var
  ar1: array[0..2] of Integer;
  ar2: array of Integer;
  ar3: Variant;

SetLength(ar2, 3);
ar3 := VarArrayCreate([0, 2], varInteger);
ar1[0] := 1;
ar2[0] := 1;
ar3[0] := 1;
```

# Палитра компонент FastScript

После установки FastScript, в Delphi / C++ Builder появится еще одна дополнительная закладка "FastScript". Эта закладка содержит все компоненты FastScript, такие как TfsScript, TfsClassesRTTI, TfsSyntaxMemo, итд.



# TfsScript - компонент для исполнения скриптов



Данный компонент является основным для использования FastScript в Вашей программе. Положите его на форму.

## Свойства:

```
SyntaxType: String;
```

Тип синтаксиса исполняемого скрипта. В поставке по умолчанию поддерживаются два вида скриптов "PascalScript" и "C++Script". По мере пополнения базы поддерживаемых языков, возможно будет указание других значений определенных в описании языка. Будьте внимательны! Свойство имеет строковый тип и легко допустить ошибку в указании типа синтаксиса. Значение по умолчанию "PascalScript".

```
Lines: TStrings;
```

Непосредственно текст скрипта. Содержит строки, предназначенные для последующего исполнения. Синтаксис должен быть в строгом соответствии с правилами написания скрипта объявленном в свойстве SyntaxType.

## Методы:

```
function Compile: Boolean;
```

Выполняет синтаксический разбор и компилирует скрипт, содержащийся в Lines, в соответствии с типом синтаксиса скрипта SyntaxType. Возвращает true, если компиляция прошла успешно, false в противном случае.

```
procedure Execute;
```

Выполняет скомпилированный ранее скрипт с помощью функции Compile.

```
function Run: boolean;
```

Компилирует и исполняет скрипт, содержащийся в Lines, в соответствии с типом синтаксиса скрипта SyntaxType. Возвращает true, если компиляция прошла успешно, false в противном случае. Применение данного метода аналогично последовательному вызову методов Compile и Execute.

## Примеры использования:

### Пример1.

Среда разработки Delphi. Загружает файл скрипта MyTestScript.pas и исполняет его. В случае ошибки, выдается сообщение.

```
fsScript1.Lines.LoadFromFile('MyTestScript.pas');
if fsScript1.Compile then
    fsScript1.Execute
else
    ShowMessage('Script compilation error!');
```

### Пример2.

Среда разработки Delphi. При нажатии на кнопку Button1, строки из компоненты fsSyntaxMemo1 присваиваются fsScript1.Lines, после чего скрипт выполняется. В случае ошибки выдается сообщение.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    fsScript1.Lines := fsSyntaxMemo1.Lines;
    if not fsScript1.Run then
        ShowMessage('Script compilation error!');
end;
```

**Пример3.** Среда разработки Delphi. Загружает файл скрипта типа "C++Script" MyTestScript.cpp и выполняет его. В случае ошибки, выдается сообщение.

```
fsScript1.Lines.LoadFromFile('MyTestScript.cpp');
fsScript1.SyntaxType := 'C++Script';
if fsScript1.Compile then
    fsScript1.Execute
else
    ShowMessage('Script compilation error!');
```

### Пример4.

Среда разработки C++Builder. Загружает файл скрипта типа "C++Script" MyTestScript.cpp и выполняет его. В случае ошибки, выдается сообщение.

```
fsScript1->Lines->LoadFromFile('MyTestScript.cpp');
fsScript1->SyntaxType = "C++Script";
if (fsScript1->Compile())
    fsScript1->Execute();
else
    ShowMessage("Script compilation error!");
```



# TfsSyntaxMemo - редактор скриптов с подсветкой синтаксиса



Усовершенствованный вариант TMemo специально адаптированный для редактирования скриптов FastScript с различным типом синтаксиса. Осуществляет подсветку следующих фрагментов текста: комментарии, зарезервированные слова, строковые значения.

Бонусный компонент, в настоящий момент не поддерживается.

**Свойства:** `SyntaxType: TSyntaxType;`

Тип подсветки синтаксиса.

Возможные значения:

**stPascal** - для языка Pascal

**stCpp** - для языка C++

**stSQL** - для языка запросов SQL

**stText** - простой текст (отключение подсветки).

Значение по умолчанию **stPascal**.

`Lines: TString;`

Редактируемый текст.

`ShowFooter: Boolean;`

Включает показ информационного поля в нижней части редактора текста, отображающего позицию курсора и пр.

`ShowGutter: Boolean;`

Включает показ информационного поля в левой части редактора текста, отображающего закладки, позицию отладчика и пр.

`BlockColor: TColor;`

Атрибуты цвета выделенного блока текста (цвета фона).

`BlockFontColor: TColor;`

Атрибуты цвета выделенного текста (цвета шрифта).

`CommentAttr: TFont;`

Атрибуты шрифта комментария.

`KeywordAttr: TFont;`

Атрибуты шрифта ключевых слов.

```
StringAttr: TFont;
```

Атрибуты шрифта строковых значений.

```
TextAttr: TFont;
```

Атрибуты шрифта простого текста.

```
Modified: Boolean;
```

True если производилось редактирование текста.

```
SelText: String;
```

Содержит выделенный текст.

### Методы:

```
procedure CopyToClipboard;
```

Копирует выделенный текст в буфер обмена.

```
procedure CutToClipboard;
```

Перемещает выделенный текст в буфер обмена.

```
procedure PasteFromClipboard;
```

Вставляет текст в позицию курсора из буфера обмена.

```
procedure SetPos(x, y: Integer);
```

Устанавливает позицию курсора в тексте. Нумерация строк и позиций начинается с 0. См. метод GetPos.

```
function GetPos: TPoint;
```

Возвращает текущую позицию курсора в тексте. См. метод SetPos.

```
procedure ShowMessage(s: String);
```

Выводит сообщение s в нижней части окна редактирования. Сообщение пропадает после любого изменения позиции курсора.

```
procedure Undo;
```

Отменяет последнее изменение.

```
function Find(Text: String): boolean;
```

Осуществляет поиск в тексте с текущей позиции курсора..

```
function IsBookmark(Line : integer): integer;
```

Возвращает номер закладки для строки с номером Line. Если закладка не установлена возвращает -1. См. метод AddBookmark.

```
procedure AddBookmark(Line, Number : integer);
```

Добавляет закладку для строки Line с номером Number. Всего поддерживается 10 закладок с номерами от 0 до 9. См. методы DeleteBookmark, GotoBookmark.

```
procedure DeleteBookmark(Number : integer);
```

Удаляет закладку с номером Number. См. метод AddBookmark.

```
procedure GotoBookmark(Number : integer);
```

Устанавливает позицию курсора на строку с закладной под номером Number. См. метод AddBookmark.

```
procedure SetActiveLine(Line : Integer);
```

Установка индикации активной строки (для применения совместно с отладчиком) в левом информационном поле редактора. Line - номер активной строки. Индикация отключается, ели Line будет равен -1. См. метод GetActiveLine.

```
function GetActiveLine: Integer;
```

Возвращает номер активной строки. Если активная строка не задана, возвращается -1. См. метод SetActiveLine.

### Клавиши редактирования.

Key	Action
Стрелки курсора	Перемещение курсора
PgUp, PgDn,	Переход на предыдущую/последующую страницу

Key	Action
<b>Ctrl+PgUp</b>	Переход в начало текста
<b>Ctrl+PgDn</b>	Переход в конец текста
<b>Home</b>	Переход в начало строки
<b>End</b>	Переход в конец строки
<b>Enter</b>	Переход на следующую строку
<b>Delete</b>	Удаление символа в позиции курсора, удаление выделенного текста
<b>Backspace</b>	Удаление символа слева от курсора
<b>Ctrl+Y</b>	Удаление текущей строки
<b>Ctrl+Z</b>	Отмена последнего изменения (до 32 событий)
<b>Shift+Arrows</b>	Выделение блока текста
<b>Ctrl+A</b>	Выделить весь текст
<b>Ctrl+U</b>	Сдвиг выделенного блока на 2 символа влево
<b>Ctrl+I</b>	Сдвиг выделенного блока на 2 символа вправо
<b>Ctrl+C, Ctrl+Insert</b>	Копирование выделенного блока в буфер обмена
<b>Ctrl+V, Shift+Insert</b>	Вставка текста из буфера обмена
<b>Ctrl+X, Shift+Delete</b>	Перенос выделенного блока в буфер обмена
<b>Ctrl+Shift+Number</b>	Установка закладки с номером 0..9 на текущей строке
<b>Ctrl+Number</b>	Переход на установленную закладку
<b>Ctrl+F</b>	Поиск строки (независимый от регистра) с позиции курсора
<b>F3</b>	Повторный поиск строки с позиции курсора

Объект класса TfsSyntaxMemo по умолчанию поддерживает операции перетаскивания текста (Drag'n'Drop) из объектов классов TTreeView, для совместного использования с объектами TfsTree.

# TfsTree - дерево функций и классов



Отображает все доступные на данный момент функции и классы в виде дерева. Бонусный компонент, в настоящий момент не поддерживается.

## Свойства:

```
property Script: TfsScript;
```

Ссылка на объект класса TfsScript.

```
property SyntaxMemo: TfsSyntaxMemo;
```

Ссылка на редактор скрипта.

```
property ShowClasses: Boolean;
```

Если true - отображается дерево классов.

```
property ShowFunctions: Boolean;
```

Если true - отображается дерево функций.

```
property Expanded: Boolean;
```

Если true - все узлы дерева отображаются в развернутом виде.

```
property ExpandLevel: integer;
```

Уровень развернутых узлов дерева. По умолчанию 2.

При двойном щелчке, если определено поле SyntaxMemo текущая запись дерева помещается в редактор скрипта. Также поддерживается перетаскивание мышью (Drag-n-drop) записей дерева в редактор TfsSyntaxMemo.

# TfsClassesRTTI компонент



Используйте этот компонент для доступа к Classes.pas из вашего приложения. Вы получаете доступ из скрипта к следующим классам:

```
TObject
constructor TObject.Create;
procedure TObject.Free;

TPersistent
procedure TPersistent.Assign(Source: TPersistent);

TList
function TList.Add(Item: TObject): Integer;
procedure TList.Clear;
procedure TList.Delete(Index: Integer);
function TList.IndexOf(Item: TObject): Integer;
procedure TList.Insert(Index: Integer; Item: TObject);
function TList.Remove(Item: TObject): Integer;
property TList.Count;
property TList.Items;

TStrings
function TStrings.Add(const S: string): Integer;
function TStrings.AddObject(const S: string; AObject: TObject): Integer;
procedure TStrings.Clear;
procedure TStrings.Delete(Index: Integer);
function TStrings.IndexOf(const S: string): Integer;
function TStrings.IndexOfName(const Name: string): Integer;
function TStrings.IndexOfObject(AObject: TObject): Integer;
procedure TStrings.Insert(Index: Integer; const S: string);
procedure TStrings.InsertObject(Index: Integer; const S: string; AObject: TObject);
procedure TStrings.LoadFromFile(const FileName: string);
procedure TStrings.LoadFromStream(Stream: TStream);
procedure TStrings.SaveToFile(const FileName: string);
procedure TStrings.SaveToStream(Stream: TStream);
property TStrings.CommaText;
property TStrings.Count;
property TStrings.Names;
property TStrings.Objects;
property TStrings.Values;
property TStrings.Strings;
property TStrings.Text;

TStringList
function TStringList.Find(s: String; var Index: Integer): Boolean;
procedure TStringList.Sort;
property TStringList.Duplicates;
property TStringList.Sorted;

TStream
function TStream.Read(Buffer: string; Count: Longint): Longint;
function TStream.Write(Buffer: string; Count: Longint): Longint;
function TStream.Seek(Offset: Longint; Origin: Word): Longint;
function TStream.CopyFrom(Source: TStream; Count: Longint): Longint;
property TStream.Position;
property TStream.Size;

TFileStream
constructor TFileStream.Create(Filename: String; Mode: Word);
```

```

TMemoryStream
procedure TMemoryStream.Clear;
procedure TMemoryStream.LoadFromStream(Stream: TStream);
procedure TMemoryStream.LoadFromFile(Filename: String);
procedure TMemoryStream.SaveToStream(Stream: TStream);
procedure TMemoryStream.SaveToFile(Filename: String);

TComponent
constructor TComponent.Create(AOwner: TComponent);
property TComponent.Owner;

TfsXMLItem
constructor TfsXMLItem.Create;
procedure TfsXMLItem.AddItem(Item: TfsXMLItem);
procedure TfsXMLItem.Clear;
procedure TfsXMLItem.InsertItem(Index: Integer; Item: TfsXMLItem);
function TfsXMLItem.Add: TfsXMLItem;
function TfsXMLItem.Find(const Name: String): Integer;
function TfsXMLItem.FindItem(const Name: String): TfsXMLItem;
function TfsXMLItem.Prop(const Name: String): String;
function TfsXMLItem.Root: TfsXMLItem;
property TfsXMLItem.Data;
property TfsXMLItem.Count;
property TfsXMLItem.Items;
property TfsXMLItem.Name;
property TfsXMLItem.Parent;
property TfsXMLItem.Text;

TfsXMLDocument
constructor TfsXMLDocument.Create;
procedure TfsXMLDocument.SaveToStream(Stream: TStream);
procedure TfsXMLDocument.LoadFromStream(Stream: TStream);
procedure TfsXMLDocument.SaveToFile(const FileName: String);
procedure TfsXMLDocument.LoadFromFile(const FileName: String);
property TfsXMLDocument.Root;

const fmCreate
const fmOpenRead
const fmOpenWrite
const fmOpenReadWrite
const fmShareExclusive
const fmShareDenyWrite
const fmShareDenyNone
const soFromBeginning
const soFromCurrent
const soFromEnd
type TDuplicates

```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iClassesRTTI" в секцию uses.

# TfsGraphicsRTTI компонент



Используйте этот компонент для доступа из скрипта к Graphics.pas. Доступны следующие классы:

```
TFont
TPen
TBrush
TCanvas
procedure TCanvas.Draw(X, Y: Integer; Graphic: TGraphic);
procedure TCanvas.Ellipse(X1, Y1, X2, Y2: Integer);
procedure TCanvas.LineTo(X, Y: Integer);
procedure TCanvas.MoveTo(X, Y: Integer);
procedure TCanvas.Rectangle(X1, Y1, X2, Y2: Integer);
procedure TCanvas.RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);
procedure TCanvas.StretchDraw(X1, Y1, X2, Y2: Integer; Graphic: TGraphic);
function TCanvas.TextHeight(const Text: string): Integer;
procedure TCanvas.TextOut(X, Y: Integer; const Text: string);
function TCanvas.TextWidth(const Text: string): Integer;
property TCanvas.Pixels;

TGraphic
procedure TGraphic.LoadFromFile(const Filename: string);
procedure TGraphic.SaveToFile(const Filename: string);
property TGraphic.Height;
property TGraphic.Width;

TMetafile
TMetafileCanvas
TBitmap
property TBitmap.Canvas

type TFontStyles
type TFontPitch
type TPenStyle
type TPenMode
type TBrushStyle
```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iGraphicsRTTI" в секцию uses.



# TfsFormsRTTI компонент



Используйте этот компонент для доступа из скрипта к StdCtrls.pas и Forms.pas. Доступны следующие классы:

```
TControl
property TControl.Parent;
procedure TControl.Hide;
procedure TControl.Show;
procedure TControl.SetBounds(ALeft, ATop, AWidth, AHeight: Integer);
event TControl.OnCanResize;
event TControl.OnClick;
event TControl.OnDblClick;
event TControl.OnMouseDown;
event TControl.OnMouseMove;
event TControl.OnMouseUp;
event TControl.OnResize;

TWinControl
procedure TWinControl.SetFocus;
event TWinControl.OnEnter;
event TWinControl.OnExit;
event TWinControl.OnKeyDown;
event TWinControl.OnKeyPress;
event TWinControl.OnKeyUp;

TCustomControl
TGraphicControl
TGroupBox
TLabel
TEdit
TMemo

TCustomComboBox
property TCustomComboBox.DroppedDown
property TCustomComboBox.ItemIndex

TComboBox
TButton
TCheckBox
TRadioButton

TCustomListBox
property TCustomListBox.ItemIndex
property TCustomListBox.SelCount
property TCustomListBox.Selected

TListBox
TControlScrollBar
TScrollingWinControl
TScrollBar

TCustomForm
procedure TCustomForm.Close;
procedure TCustomForm.Hide;
procedure TCustomForm.Show;
function TCustomForm.ShowModal: Integer;
event TCustomForm.OnActivate
event TCustomForm.OnClose
event TCustomForm.OnCloseQuery
event TCustomForm.OnCreate
event TCustomForm.OnDestroy
```

```
event TCustomForm.OnDestroy  
event TCustomForm.OnDeactivate  
event TCustomForm.OnHide  
event TCustomForm.OnPaint  
event TCustomForm.OnShow  
property TCustomForm.ModalResult
```

TForm

```
type TModalResult  
type TCursor  
type TShiftState  
type TAlignment  
type TAlign  
type TMouseButton  
type TAnchors  
type TBevelCut  
type TTextLayout  
type TEditCharCase  
type TScrollStyle  
type TComboBoxStyle  
type TCheckBoxState  
type TListBoxStyle  
type TFormBorderStyle  
type TWindowState  
type TFormStyle  
type TBorderIcons  
type TPosition  
type TCloseAction
```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iFormsRTTI" в секцию uses.

# TfsExtCtrlsRTTI компонент



Используйте этот компонент для доступа из скрипта к ExtCtrls.pas. Доступны следующие классы:

```
TShape

TPaintBox
event TPaintBox.OnPaint

TImage
TBevel

TTimer
event TTimer.OnTimer

TPanel
TSplitter
TBitBtn
TSpeedButton

TCheckListBox
property TCheckListBox.Checked

TTabControl
TTabSheet

TPageControl
procedure TPageControl.SelectNextPage(GoForward: Boolean);
property TPageControl.PageCount;
property TPageControl.Pages;

TStatusPanel

TStatusPanels
function TStatusPanels.Add: TStatusPanel
property TStatusPanels.Items

TStatusBar

TTreeNode
procedure TTreeNode.Delete;
function TTreeNode.EditText: Boolean;
property TTreeNode.Count
property TTreeNode.Data
property TTreeNode.ImageIndex
property TTreeNode.SelectedIndex
property TTreeNode.StateIndex
property TTreeNode.Text

TTreeNodees
function TTreeNodees.Add(Node: TTreeNode; const S: string): TTreeNode;
function TTreeNodees.AddChild(Node: TTreeNode; const S: string): TTreeNode;
procedure TTreeNodees.BeginUpdate;
procedure TTreeNodees.Clear;
procedure TTreeNodees.Delete(Node: TTreeNode);
procedure TTreeNodees.EndUpdate;
property TTreeNodees.Count;
property TTreeNodees.Item;

TTreeView
procedure TTreeView.FullCollapse;
```

```

procedure TTreeView.FullCollapse;
procedure TTreeView.FullExpand;
property TTreeView.Selected;
property TTreeView.TopItem;

TTrackBar
TProgressBar
TListColumn

TListColumns
function TListColumns.Add: TListColumn
property TListColumns.Items

TListItem
procedure TListItem.Delete;
function TListItem.EditCaption: Boolean;
property TListItem.Caption
property TListItem.Checked
property TListItem.Data
property TListItem.ImageIndex
property TListItem.Selected
property TListItem.StateIndex
property TListItem.SubItems

TListItems
function TListItems.Add: TListItem;
procedure TListItems.BeginUpdate;
procedure TListItems.Clear;
procedure TListItems.Delete(Index: Integer);
procedure TListItems.EndUpdate;
property TListItems.Count
property TListItems.Item

TIconOptions
TListView
TToolButton
TToolBar
TMonthCalColors
TDateTimePicker
TMonthCalendar

type TShapeType
type TBevelStyle
type TBevelShape
type TResizeStyle
type TButtonLayout
type TButtonState
type TButtonStyle
type TBitBtnKind
type TNumGlyphs
type TTabPosition
type TTabStyle
type TStatusPanelStyle
type TStatusPanelBevel
type TSortType
type TTrackBarOrientation
type TTickMark
type TTickStyle
type TProgressBarOrientation
type TIconArrangement
type TListArrangement
type TViewStyle
type TToolButtonStyle
type TDateTimeKind
type TDTDateMode
type TDTDateFormat
type TDTCalAlignment
type TCalDayOfWeek

```

---

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iExtCtrlsRTTI" в секцию uses.

# TfsDialogsRTTI компонент



Используйте этот компонент для доступа из скрипта к Dialogs.pas. Доступны следующие классы:

```
TCommonDialog
function TCommonDialog.Execute: Boolean;
TOpenDialog
TSaveDialog
TColorDialog
TFontDialog
TPrintDialog
TPrinterSetupDialog

type TOpenOptions
type TFileEditStyle
type TColorDialogOptions
type TFontDialogOptions
type TFontDialogDevice
type TPrintRange
type TPrintDialogOptions
```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iDialogsRTTI" в секцию uses.

# TfsDBRTTI компонент



Используйте этот компонент для доступа из скрипта к DB.pas. Доступны следующие классы:

```
TField
property TField.AsBoolean
property TField.AsCurrency
property TField.AsDateTime
property TField.AsFloat
property TField.AsInteger
property TField.AsDate
property TField.AsTime
property TField.AsString
property TField.AsVariant
property TField.DataType
property TField.DisplayName
property TField.DisplayText
property TField.IsNull
property TField.Size
property TField.Value

TFields
property TFields.Fields

TStringField
TNumericField
TIntegerField
TSmallIntField
TWordField
TAutoIncField
TFloatField
TCurrencyField
TBooleanField
TDateTimeField
TDateField
TTimeField
TBinaryField
TBytesField
TVarBytesField
TBCDField

TBlobField
procedure TBlobField.LoadFromFile(const FileName: String);
procedure TBlobField.LoadFromStream(Stream: TStream);
procedure TBlobField.SaveToFile(const FileName: String);
procedure TBlobField.SaveToStream(Stream: TStream);

TMemoField
TGraphicField
TFieldDef
TFieldDefs
property TFieldDefs.Items

TDataSource
type TBookmark

TDataSet
procedure TDataSet.Open;
procedure TDataSet.Close;
procedure TDataSet.First;
procedure TDataSet.Last;
```

```

procedure TDataSet.Last;
procedure TDataSet.Next;
procedure TDataSet.Prior;
procedure TDataSet.Cancel;
procedure TDataSet.Delete;
procedure TDataSet.Post;
procedure TDataSet.Append;
procedure TDataSet.Insert;
procedure TDataSet.Edit;
function TDataSet.FieldByName(const FieldName: string): TField;
procedure TDataSet.GetFieldNames(List: TStrings);
function TDataSet.FindFirst: Boolean;
function TDataSet.FindLast: Boolean;
function TDataSet.FindNext: Boolean;
function TDataSet.FindPrior: Boolean;
procedure TDataSet.FreeBookmark(Bookmark: TBookmark);
function TDataSet.GetBookmark: TBookmark;
procedure TDataSet.GotoBookmark(Bookmark: TBookmark);
function TDataSet.Locate(const KeyFields: string; const KeyValues: Variant; Options: TLocateOptions): Boolean;
function TDataSet.IsEmpty: Boolean;
property TDataSet.Bof
property TDataSet.Eof
property TDataSet.FieldCount
property TDataSet.FieldDefs
property TDataSet.Fields
property TDataSet.Filter
property TDataSet.Filtered
property TDataSet.FilterOptions
property TDataSet.Active

TParam
procedure TParam.Clear;
property TParam.Bound
property TParam.IsNull
property TParam.Text
property TParam.AsBoolean
property TParam.AsCurrency
property TParam.AsDateTime
property TParam.AsFloat
property TParam.AsInteger
property TParam.AsDate
property TParam.AsTime
property TParam.AsString
property TParam.AsVariant

TParams
function TParams.ParamByName(const Value: string): TParam;
function TParams.FindParam(const Value: string): TParam;
property TParams.Items

type TFieldType
type TBlobStreamMode
type TLocateOptions
type TFilterOptions
type TParamType

```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iDBRTTI" в секцию uses.



# TfsDBCtrlsRTTI компонент



Используйте этот компонент для доступа из скрипта к DBCtrls.pas. Доступны следующие классы:

```
TDBEdit
TDBText
TDBCheckBox
property TDBCheckBox.Checked
TDBComboBox
property TDBComboBox.Text
TDBListBox
TDBRadioGroup
property TDBRadioGroup.ItemIndex
property TDBRadioGroup.Value
TDBMemo
TDBImage
TDBNavigator
TDBLookupControl
property TDBLookupControl.KeyValue
TDBLookupListBox
property TDBLookupListBox.SelectedItem
TDBLookupComboBox
property TDBLookupComboBox.Text
TColumnTitle
TColumn
TDBGridColumns
function TDBGridColumns.Add: TColumn;
property TDBGridColumns.Items
TDBGrid

type TButtonSet
type TColumnButtonStyle
type TDBGridOptions
```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iDBCtrlsRTTI" в секцию uses.

# TfsBDERTTI компонент



Используйте этот компонент для доступа из скрипта к BDE. Доступны следующие классы:

```
TSession
TDatabase
TBDEDataSet
TDBDataSet
TTable
procedure TTable.CreateTable;
procedure TTable.DeleteTable;
procedure TTable.EmptyTable;
function TTable.FindKey(const KeyValues: array): Boolean;
procedure TTable.FindNearest(const KeyValues: array);
procedure TTable.RenameTable(const NewTableName: string);
TQuery
procedure TQuery.ExecSQL;
function TQuery.ParamByName(const Value: string): TParam;
procedure TQuery.Prepare;
property TQuery.ParamCount;
TStoredProc
procedure TStoredProc.ExecProc;
function TStoredProc.ParamByName(const Value: string): TParam;
procedure TStoredProc.Prepare;
property TStoredProc.ParamCount;
type TTableType
type TParamBindMode
```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iBDERTTI" в секцию uses.

# TfsADORTTI компонент



Используйте этот компонент для доступа из скрипта к ADO. Доступны следующие классы:

```
TADOConnection
TParameter
TParameters
property TParameters.Items
TCustomADODataset
TADOTable
TADOQuery
procedure TADOQuery.ExecSQL;
TADOStoredProc
procedure TADOStoredProc.ExecProc;
type TDataType
```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iADORTTI" в секцию uses.

# TfsIBXRTTI компонент



Используйте этот компонент для доступа из скрипта к IBX. Доступны следующие классы:

```
TIBDataBase  
TIBTransaction  
TIBCustomDataSet  
TIBTable  
TIBQuery  
procedure TIBQuery.ExecSQL;  
TIBStoredProc  
procedure TIBStoredProc.ExecProc;
```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iIBXRTTI" в секцию uses.

# TfsChartRTTI компонент



Используйте этот компонент для доступа из скрипта к TeeChart. Доступны следующие классы:

```
TChartValueList
TChartAxisTitle
TChartAxis
TCustomChartLegend
TChartLegend
TSeriesMarks
TChartGradient
TChartWall
TChartBrush
TChartTitle
TChartSeries
procedure TChartSeries.Clear;
procedure TChartSeries.Add(const AValue: Double; const ALabel: String; AColor: TColor);
TSeriesPointer
TCustomSeries
TLineSeries
TPointSeries
TAreaSeries
TCustomBarSeries
TBarSeries
THorizBarSeries
TCircledSeries
TPieSeries
TFastLineSeries
TCustomChart
TChart
type TChartValue
type TLegendStyle
type TLegendAlignment
type TLegendTextStyle
type TChartListOrder
type TGradientDirection
type TSeriesMarksStyle
type TAxisLabelStyle
type THorizAxis
type TVertAxis
type TTeeBackImageMode
type TPanningMode
type TSeriesPointerStyle
type TMultiArea
type TMultiBar
type TBarStyle
```

Вы получаете доступ ко всем published свойствам этих классов + доступ к некоторым public свойствам и методам.

Примечание: это «фиктивный» компонент. Он нужен только для автоматического включения модуля "FS\_iChartRTTI" в секцию uses.

# Использование скриптов

# Простейший пример использования скрипта

Этот пример демонстрирует простейший способ использования FastScript. Положите на форму компоненты TfsScript и TButton. Щелкните мышью на созданной кнопке. Создайте обработчик события Button1Click :

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  fsScript1.Clear;
  fsScript1.Lines.Text := 'begin ShowMessage(''Hello!'') end.';
  fsScript1.Parent := fsGlobalUnit;
  fsScript1.SyntaxType := 'PascalScript';
  if not fsScript1.Run then
    ShowMessage(fsScript1.ErrorMsg);
end;
```

- Если компонент используется для запуска нескольких скриптов, надо вызвать метод Clear;
- В свойство Lines компоненты fsScript1 мы помещаем текст скрипта;
- Свойство Parent подключаем к глобальному модулю (он содержит объявления стандартных классов и функций);
- Запускаем скрипт на исполнение с помощью команды fsScript1.Run;
- Используется тип синтаксиса PascalScript. Если во время компиляции возникли ошибки выдаем сообщение с текстом ошибки fsScript1.ErrorMsg.

Альтернативный способ, без использования fsGlobalUnit (для корректной работы в многопоточных приложениях):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  fsScript1.Clear;
  fsScript1.AddRTTI;
  fsScript1.Lines.Text := 'begin ShowMessage(''Hello!'') end.';
  fsScript1.SyntaxType := 'PascalScript';
  if not fsScript1.Run then
    ShowMessage(fsScript1.ErrorMsg);
end;
```

# Получение списка поддерживаемых языков

Для получения списка поддерживаемых языков, нужно вызвать процедуру `fsGetLanguageList(list: TStrings)`, определенную в модуле `FS_iTools`.

```
uses FS_iTools;  
  
fsGetLanguageList(LangComboBox.Items);
```



# Получение подробной информации о синтаксической ошибке

```
uses FS_iInterpreter, FS_iTools;

begin
  if not fsScript1.Compile then
    begin
      { show the error message and position in the status bar }
      StatusBar1.Text := fsScript1.ErrorMessage + ' at ' + fsScript1.ErrorPos;
      Exit;
    end
  else
    fsScript1.Execute;
  end;
end;
```

# Отладка скрипта

Используйте OnRunLine. Например:

```
procedure TForm1.OnRunLine(Sender: TfsScript; const UnitName, SourcePos: String);
var
  pt: TPoint;
begin
  // locate the unit with UnitName name
  ...
  // locate the line with pt.Y number
  pt := fsPosToPoint(SourcePos);

  FStopped := True;
  while FStopped do
    Application.ProcessMessages;
end;
```

Также смотрите демонстрационный пример в папке DEMOS\Main.

# Добавление процедуры в скрипт

Для добавления процедуры/функции в скрипт выполните следующие действия:

- Создайте обработчик - функцию TfsCallMethodEvent.
- Вызовите метод TfsScript.AddMethod. Первый параметр - это синтаксис функции (обратите внимание - синтаксис, независимо от используемого вами языка, должен быть паскалевским!), второй - ссылка на обработчик TfsCallMethodEvent.

```
{ собственно, функция }
procedure TForm1.DelphiFunc(s: String; i: Integer);
begin
  ShowMessage(s + ', ' + IntToStr(i));
end;

{ обработчик TfsCallMethodEvent }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String;
  var Params: Variant): Variant;
begin
  DelphiFunc(Params[0], Params[1]);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  { clear all items }
  fsScript1.Clear;
  { script text }
  fsScript1.Lines := Memo1.Lines;
  { frGlobalUnit contains standard types and functions }
  fsScript1.Parent := fsGlobalUnit;
  { делаем процедуру DelphiFunc доступной в скрипте }
  fsScript1.AddMethod('procedure DelphiFunc(s: String; i: Integer)', CallMethod);

  { компилируем скрипт с использованием PascalScript }
  if fsScript1.Compile then
    fsScript1.Execute else { выполнить, если компиляция успешна }
    ShowMessage(fsScript1.ErrorMessage); { вывести сообщение об ошибке }
end;
```

Если вы желаете добавить несколько методов, вы можете сделать это, используя один дескриптор метода:

```
Prog.AddMethod('procedure DelphiFunc(s: String; i: Integer)', CallMethod);
Prog.AddMethod('procedure DelphiFunc2(s: String)', CallMethod);

{ дескриптор метода }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String;
  var Params: Variant): Variant;
begin
  { определение вызываемого метода }
  if MethodName = 'DELPHIFUNC' then
    DelphiFunc(Params[0], Params[1])
  else if MethodName = 'DELPHIFUNC2' then
    DelphiFunc2(Params[0]);
end;
```

# Добавление функции в скрипт

Так же, как и добавление процедуры.

```
fsScript1.AddMethod('function DelphiFunc2(s: String): Boolean', CallMethod);

{ обработчик }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String;
  var Params: Variant): Variant;
begin
  Result := DelphiFunc(Params[0]);
end;
```

# Добавление функции с var and параметрами по умолчанию

Вам не нужно заботиться о параметрах по умолчанию - FastScript подставит их автоматически. Параметры var должны обрабатываться вами.

```
fsScript1.AddMethod('function DelphiFunc(var s: String; i: Integer = 0): Boolean', CallMethod);

{ обработчик }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String;
  var Params: Variant): Variant;
var
  s: String;
begin
  s := Params[0];
  Result := DelphiFunc(s, Params[1]);
  Params[0] := s;
end;
```

# Добавление функции с параметром class

Поскольку все параметры представляются как массив типа Variant, вам надо преобразовать их в объекты.

```
Prog.AddMethod('procedure HideButton(Button: TButton)', CallMethod);

{ обработчик }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String;
    var Params: Variant): Variant;
begin
    TButton(Integer(Params[0])).Hide;
end;
```

# Добавление функции, возвращающей значение типа class

Поскольку значения, возвращаемые дескриптором метода, это массив типа Variant, вам надо преобразовать результаты типа TObject к Variant.

```
fsScript1.AddMethod('function MainForm: TForm', CallMethod);

{ обработчик }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String;
  var Params: Variant): Variant;
begin
  Result := Integer(Form1);
end;
```

## Добавление константы в скрипт

Для добавления в скрипт константы вызовите метод `TfsScript.AddConst`. Первый параметр - это наименование константы, второй - тип (должен быть одним из стандартных типов), третий - значение.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { добавление в скрипт константы }
  fsScript1.AddConst('pi', 'Extended', 3.14159);

  { компилируем скрипт с использованием языка PascalScript }
  fsScript1.Lines := Memo1.Lines;
  fsScript1.SyntaxType := 'PascalScript';
  fsScript1.Parent := fsGlobalUnit;
  if fsScript1.Compile then
    fsScript1.Execute else { выполнить, если компиляция прошла успешно }
    ShowMessage(fsScript1.ErrorMessage); { вывести сообщение об ошибке }
end;
```



## Добавление переменной в скрипт

Для добавления в скрипт переменной вызовите метод `TfsScript.AddVariable`. Этот метод подобен методу `AddConst`, за исключением того, что вы можете изменить значение переменной в скрипте. Обратите внимание, что переменная Delphi не изменится после выполнения скрипта.

```
fsScript1.AddVariable('i', 'Integer', i);
```

## Добавление объекта в скрипт

Для добавления объекта в скрипт вызовите метод `TfsScript.AddObject`. Первый параметр это имя объекта, второй - собственно, объект.

```
fsScript1.AddObject('Button1', Button1);
```

Если добавляемый объект имеет незарегистрированный класс, то предварительно надо зарегистрировать его:

```
fsScript1.AddClass(TForm1, 'TForm1');  
fsScript1.AddObject('Form1', Form1);
```

Вы также можете использовать метод `fsGlobalUnit.AddForm` для добавления формы или модуля данных вместе со всеми дочерними компонентами:

```
fsGlobalUnit.AddForm(Form1);
```

В этом случае регистрировать класс формы с помощью `AddClass` не требуется. Теперь вы можете обращаться к элементам формы из скрипта:

```
Form1.Button1.Caption := '...'
```

## Добавления описания типа в скрипт

Для добавления собственного типа в скрипт вызовите метод `TfsScript.AddType`. первый параметр это наименование типа, второй - один из следующих поддерживаемых типов:

```
TfsVarType = (fvtInt, fvtBool, fvtFloat, fvtChar, fvtString, fvtClass, fvtArray, fvtVariant, fvtEnum);  
  
fsScript1.AddType('TCursor', fvtInt);
```

## Добавление перечисления в скрипт

Для добавления перечислимого типа в скрипт вызовите метод `TfsScript.AddEnum`. Первый параметр это имя типа, второй - значения, разделенные запятыми.

```
fsScript1.AddEnum('TPrinterOrientation', 'poPortrait, poLandscape');
```

## Добавление множества в скрипт

Для добавления множественного в скрипт вызовите метод `TfsScript.AddEnumSet`. Первый параметр это наименование типа, второй - значения, разделенные запятыми.

```
fsScript1.AddEnumSet('TFontStyles', 'fsBold, fsItalic, fsUnderline, fsStrikeOut');
```

## Добавление класса в скрипт

Для добавления в скрипт класса вызовите метод `TfsScript.AddClass`. Первый параметр это наименование класса, второй - имя базового класса. Например:

```
type
  TMyClass = class(TObject)
  ...
end;

fsScript1.AddClass(TMyClass, 'TObject');
```

Это сделает все `published` свойства данного класса доступными. Если вы желаете сделать этот класс доступным для всех скриптов, рекомендуется добавить этот класс в `fsGlobalUnit`, который является предком всех скриптов.

# Реализация public свойств и методов класса

Метод AddClass автоматически добавляет все published свойства класса. Public свойства и методы требуют дополнительной работы. Следующий пример демонстрирует как добавить public-метод в класс. Вам потребуется создать дескриптор метода (функция типа TfsCallMethod).

```
begin
  ...
  { добавим новый класс, унаследованный от TObject }
  with fsScript1.AddClass(TList, 'TObject') do
  begin
    { добавим public методы }
    AddMethod('function Add(Item: TObject): Integer', CallMethod);
    AddMethod('procedure Clear', CallMethod);
  end;
  ...
end;

{ обработчик }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass;
  const MethodName: String; var Params: Variant): Variant;
begin
  Result := 0;

  if MethodName = 'ADD' then
  { преобразуем параметр типа Variant к типу Pointer и вызовем с ним метод Add }
    TList(Instance).Add(Pointer(Integer(Params[0])))
  else if MethodName = 'CLEAR' then
    TList(Instance).Clear
  end;
end;
```

Для реализации свойства вам потребуется создать дескриптор метода и два дескриптора свойства типов TfsGetValueEvent и TfsSetValueEvent:

```
TfsGetValueEvent = function(Instance: TObject; ClassType: TClass; const PropName: String): Variant of
object;
TfsSetValueEvent = procedure(Instance: TObject; ClassType: TClass; const PropName: String; Value:
Variant) of object;
```

Индексируемые (indexed) свойства и свойства по умолчанию (default) описываются дескриптором метода, обычные свойства описываются дескрипторами свойств Get/Set.

```

begin
  ...
  with fsScript1.AddClass(TStrings, 'TPersistent') do
  begin
    { property CommaText: String }
    AddProperty('CommaText', 'string', GetProp, SetProp);
    { property Count: Integer readonly, second handler is nil }
    AddProperty('Count', 'Integer', GetProp, nil);
    { index property Objects[Index: Integer]: TObject }
    AddIndexProperty('Objects', 'Integer', 'TObject', CallMethod);
    { default property Strings[Index: Integer]: String }
    AddDefaultProperty('Strings', 'Integer', 'string', CallMethod);
  end;
  ...
end;

{ обработчик }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass;
  const MethodName: String; var Params: Variant): Variant;
begin
  Result := 0;

  if MethodName = 'OBJECTS.GET' then
    Result := Integer(TStrings(Instance).Objects[Params[0]])
  else if MethodName = 'OBJECTS.SET' then
    TStrings(Instance).Objects[Params[0]] := TObject(Integer(Params[1]))
  else if MethodName = 'STRINGS.GET' then
    Result := TStrings(Instance).Strings[Params[0]]
  else if MethodName = 'STRINGS.SET' then
    TStrings(Instance).Strings[Params[0]] := Params[1]
end;

{ обработчик }
function TForm1.GetProp(Instance: TObject; ClassType: TClass;
  const PropName: String): Variant;
begin
  Result := 0;

  if PropName = 'COMMA TEXT' then
    Result := TStrings(Instance).CommaText
  else if PropName = 'COUNT' then
    Result := TStrings(Instance).Count
end;

{ обработчик }
procedure TForm1.SetProp(Instance: TObject; ClassType: TClass;
  const PropName: String; Value: Variant);
begin
  if PropName = 'COMMA TEXT' then
    TStrings(Instance).CommaText := Value
end;

```



# Реализация обработчика события класса

Для добавления события в класс используйте метод `TfsClassVariable.AddEvent`. Первый параметр это наименование события, второй - описатель события.

```
with fsScript1.AddClass(TControl, 'TComponent') do  
    AddEvent('OnClick', TfsNotifyEvent);
```

Некоторые predefined в модуле `FS_iEvents` описатели событий:

```
TfsNotifyEvent  
TfsMouseEvent  
TfsMouseMoveEvent  
TfsKeyEvent  
TfsKeyPressEvent  
TfsCloseEvent  
TfsCloseQueryEvent  
TfsCanResizeEvent
```

Список описателей событий, доступных в скрипте, см. в разделах "`TfsFormsRTTI` ", "`TfsExtCtrlsRTTI` " и "`TfsDBCtrlsRTTI` ".

# Реализация нестандартных обработчиков событий

Вот некоторые predefined в модуле FS\_iEvents описатели событий:

```
TfsNotifyEvent  
TfsMouseEvent  
TfsMouseMoveEvent  
TfsKeyEvent  
TfsKeyPressEvent  
TfsCloseEvent  
TfsCloseQueryEvent  
TfsCanResizeEvent
```

Однако, однако, если вы желаете создать собственный обработчик события, посмотрите на следующий пример:

```

{ пример двух описателей событий }
type
  { аналог TNotifyEvent }
  TfsNotifyEvent = class(TfsCustomEvent)
  public
    procedure DoEvent(Sender: TObject);
    function GetMethod: Pointer; override;
  end;

  { аналог of TKeyPressEvent = procedure(Sender: TObject; var Key: Char) }
  TfsKeyPressEvent = class(TfsCustomEvent)
  public
    procedure DoEvent(Sender: TObject; var Key: Char);
    function GetMethod: Pointer; override;
  end;

{ TfsNotifyEvent }

procedure TfsNotifyEvent.DoEvent(Sender: TObject);
begin
  { Встроенный метод CallHandler }
  CallHandler([Sender]);
end;

function TfsNotifyEvent.GetMethod: Pointer;
begin
  Result := @TfsNotifyEvent.DoEvent;
end;

{ TfsKeyPressEvent }

procedure TfsKeyPressEvent.DoEvent(Sender: TObject; var Key: Char);
begin
  CallHandler([Sender, Key]);
  { получение параметра var }
  Key := String(Handler.Params[1].Value)[1];
end;

function TfsKeyPressEvent.GetMethod: Pointer;
begin
  Result := @TfsKeyPressEvent.DoEvent;
end;

```

# Доступ к переменным из программы

Для получения/установки значения переменных скрипта, используйте свойство `TfsScript.Variables`.

```
val := fsScript1.Variables['i'];  
fsScript1.Variables['i'] := 10;
```

# Вызов функции из программы

Для вызова скриптовой функции используйте метод `TfsScript.CallFunction`. Первый параметр это имя вызываемой функции, второй - это параметры функции.

```
// вызовет 'function ScriptFunc(s: String; i: Integer)'  
val := fsScript1.CallFunction('ScriptFunc', VarArrayOf(['hello', 1]));
```

# Вызов функции с параметром var из программы

Так же, как описано выше.

```
var
  Params: Variant;

Params := VarArrayOf(['hello', 1]);
// вызовет 'function ScriptFunc(var s: String; i: Integer)'
fsScript1.CallFunction1('ScriptFunc', Params);
ShowMessage(Params[0]);
```

# Вычисление выражений

Если вы желаете вычислить некое выражение (к примеру, 'i+1'), вызовите метод TfsScript.Evaluate.

```
ShowMessage(fsScript1.Evaluate('i+1'));
```

Это полезно для использования в отладочных целях.

# Запись и восстановление скомпилированного кода

Иногда необходимо сохранить результат компиляции и выполнить его позже. Это можно сделать с помощью методов `TfsScript.GetILCode` и `SetILCode`. Код ниже компилирует исходный скрипт и помещает результат компиляции в поток:

```
var
  s: TStream;

fsScript1.Lines.Text := ...;
fsScript1.GetILCode(s);
```

После этого, вы можете восстановить скомпилированный код из потока и выполнить его:

```
fsScript1.SetILCode(s);
fsScript1.Execute;
```



# Использование директивы "uses"

Вы можете разбивать большой скрипт на модули, подобно тому, как это делается в Object Pascal. Для использования модуля служит директива "uses". Вот пример ее применения:

Файл unit1.pas:

```
uses 'unit2.pas';

begin
  Unit2Proc('Hello!');
end.
```

Файл unit2.pas:

```
procedure Unit2Proc(s: String);
begin
  ShowMessage(s);
end;

begin
  ShowMessage('initialization of unit2...');
end.
```

Как видно, отличие от Object Pascal заключается в том, что мы указываем в uses имя файла с расширением в одинарных кавычках. Подключаемый файл должен иметь такую же структуру, как и основной. Код, заключенный в основной блок begin..end, будет выполнен при подключении модуля - это аналог секции initialization в Pascal. В этом примере нельзя использовать unit1 из unit2 - это вызовет бесконечный цикл при попытке откомпилировать такой скрипт. Подобные ссылки невозможны, т.к. в FastScript нет аналогов паскалевским конструкциям interface/implementation. Пользуясь директивой #language, можно писать многоязычные скрипты. Так, один модуль может быть написан на PascalScript, другой - на C++Script:

Файл unit1.pas:

```
uses 'unit2.pas';

begin
  Unit2Proc('Hello from PascalScript!');
end.
```

Файл unit2.pas:

```
#language C++Script

void Unit2Proc(string s)
{
    ShowMessage(s);
}

{
    ShowMessage("unit2 initialization, C++Script");
}
```

Директива #language должна быть первой строкой в файле. Если директива присутствует, она перекрывает значение TfsScript.SyntaxType.

# Обучение работе со скриптами

Обучающие примеры работы скриптов расположены в папке DEMOS\Main\Samples. Скомпилируйте демонстрационную программу из папки DEMOS\Main и откройте в ней один из примеров скриптов.