Introduction

What is FastScript

Quick start

Features and missing features

Language reference

Script structure

Data types

Classes

Functions

Events

Enumerations and sets

Arrays
**What is FastScript**

FastScript is a scripting library. It is useful for the programmers who want to add scripting ability to their projects. FastScript is written on 100% Object Pascal and can be installed in Borland Delphi 4-7, C++Builder 4-6 and Kylix 1-3. Unique feature of FastScript is ability to use several languages (PascalScript, C++Script, JScript and BasicScript), so you can write scripts using your favourite language. FastScript doesn't use Microsoft Scripting Host, so it can be used in Windows and Linux environment.

FastScript combines cross-platform scripting, fast code execution, small footprint, rich set of features and a splendid scalability. Make your applications the most flexible and powerful ones with FastScript!
Quick start

Here is a sample code which demonstrates the easiest way of using FastScript. For the correct work of the example put the components fsScript1: TfsScript and fsPascal1: TfsPascal on the form.

```pascal
uses FS_iInterpreter;

procedure TForm1.Button1Click(Sender: TObject);
begin
  fsScript1.Clear; // do this if you running many scripts from one component
  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;
```

As you can see, there is nothing difficult here. At first we fill in the fsScript1.Lines property with the script text. For using standard types and functions we set Parent property to the fsGlobalUnit. Then we compile the script using PascalScript language (you can use C++Script, BasicScript, JScript as well). If compilation is successful Compile method returns True and we can Execute the script. Otherwise the error message is shown.
Features and missing features

Features

- Multi-language architecture allows you to use a number of languages (at present moment PascalScript, C++Script, BasicScript, JScript). You can add any procedural language (language grammar is stored in XML format).
- Standard language set: variables, constants, procedures, functions (nested functions allowed) with var/const/default parameters, all the standard operators and statements (including case, try/finally/except, with), types (int, float, bool, char, string, multi-dimensional array, enum, variant), classes (with methods, events, properties, indices and default properties).
- Types compatibility checking.
- Access to any object inside your application. Standard libraries for the access to the base classes, controls, forms and BD. Easily expandable library architecture.
- Small footprint - 90-150Kb depending on used modules.
- Can be used in multi-thread environment.

Missing features

- No type declarations (records, classes) in the script; no records, no pointers, no sets (but you can use 'IN' operator - "a in ['a..'c','d']"), no shortstrings, no GOTO statement.
- C++Script: no octal constants; no 'break' in the SWITCH operator (SWITCH works like Pascal CASE); '++' and '--' operators are possible only after the variables, that is '++i' is not allowed; '---', '+++' and '=' operators do not return a value, that is 'if(i++)' is not allowed; all the identifiers are case-insensitive; NULL constant is the Pascal Null - use nil instead of NULL.
- JScript and BasicScript: see syntax diagrams.
**Language reference**

**PascalScript syntax:**

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 -> '>'
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

**C++Script syntax:**

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]/',')...

**JScript syntax:**

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/','...

Statement -> (AssignStmt | CallStmt | BreakStmt | ContinueStmt | DeleteStmt | DoWhileStmt | ForStmt | FunctionStmt | IfStmt | ImportStmt | ReturnStmt | SwitchStmt | VarStmt | WhileStmt | WithStmt | Block) [';']

BreakStmt -> BREAK
ContinueStmt -> CONTINUE
DeleteStmt -> DELETE Designator
AssignStmt -> Designator ['+'|'-'|'*'|'/''] '=' Expression
CallStmt -> Designator ['+'|'+|--'|'-']
ReturnStmt -> RETURN [Expression]
IfStmt -> IF '(' Expression ')' Statement [ELSE Statement]
SwitchStmt -> SWITCH '(' Expression ')' '{' (CaseSelector)... [DEFAULT ':' Statement] '}'
CaseSelector -> CASE SetConstructor ':' Statement
DoWhileStmt -> DO Statement [';'] WHILE '(' Expression ')' ';'
WhileStmt -> WHILE '(' Expression ')' Statement
ForStmt -> FOR '(' ForStmtItem ';' Expression ';' ForStmtItem ')'
Statement
ForStmtItem -> AssignStmt
  -> CallStmt
  -> VarStmt
  -> Empty
TryStmt -> TRY CompoundStmt (FINALLY | EXCEPT) CompoundStmt
FunctionStmt -> FunctionHeading Block
FunctionHeading -> FUNCTION Ident FormalParameters
FormalParameters -> '([' [FormalParam/',',']]
FormalParam -> ['&'] Ident
WithStmt -> WITH '(' Designator ')' Statement
BasicScript syntax:

Program -> Statements

Statements -> (EOL | StatementList EOL)...

StatementList -> Statement/':'...

ImportStmt -> IMPORTS (String/,'...')

DimStmt -> DIM (VarDecl/','...')

VarDecl -> Ident [Array] [AsClause] [InitValue]

AsClause -> AS Ident

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

ArrayDim -> 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


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

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

FormalParm -> [BYREF | BYVAL] VarList

**Script structure**

The PascalScript structure is the same as in Object Pascal language:

```pascal
#language PascalScript // this is optional
program MyProgram;    // this is optional
uses 'unit1.pas', 'unit2.pas';
// uses section - must be before any other sections
// v1.2 changes: warning! the included units are not inserted into the
// main unit text. So it can have
// 'program', 'uses' sections and must have the 'main procedure'
// section.
var
  i, j: Integer;  // var section
const
  pi = 3.14159;  // const section
procedure p1;    // procedures and function
  var
    i: Integer;
begin
  procedure p2;  // nested procedure
  begin
  end;
```
begin end;
begin // main procedure that will be executed.
end.

The C++Script structure is:

```cpp
#include "unit1.cpp", "unit2.cpp"
// uses section - must be before any other sections

int i, j; // var section
#DEFINE pi = 3.14159 // const section

void pl() // procedures and function
{
    // there is no nested procedures in C++Script
}
{
    // main procedure that will be executed.
}
```

The JScript structure is:

```javascript
import "unit1.js", "unit2.js"
// import section - must be before any other sections

var i, j = 0; // var section

function pl() // procedures and function
{
    //
}
{
    // main procedure that will be executed.
}

pl();
for (i = 0; i < 10; i++) j++;
```

The BasicScript structure is:

```basic
imports "unit1.vb", "unit2.vb"
// imports section - must be before any other sections

dim i, j = 0 // var section

function fl() // procedures and function
end function //

sub pl()
```
end sub

// main procedure that will be executed.
for i = 0 to 10
    p1()
next
**Data types**

Internally FastScript operates with the Variant type and is based on it. Nevertheless, you can use the following predetermined types in your scripts:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>Same as Integer type</td>
</tr>
<tr>
<td>Word</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>Longint</td>
<td></td>
</tr>
<tr>
<td>Cardinal</td>
<td></td>
</tr>
<tr>
<td>TColor</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean type</td>
</tr>
<tr>
<td>Real</td>
<td>Same as Extended type</td>
</tr>
<tr>
<td>Single</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td></td>
</tr>
<tr>
<td>Extended</td>
<td></td>
</tr>
<tr>
<td>TDate</td>
<td></td>
</tr>
<tr>
<td>TTime</td>
<td></td>
</tr>
<tr>
<td>TDateTime</td>
<td></td>
</tr>
<tr>
<td>Char</td>
<td>Char type</td>
</tr>
<tr>
<td>String</td>
<td>String type</td>
</tr>
<tr>
<td>Variant</td>
<td>Same as Variant type</td>
</tr>
<tr>
<td>Pointer</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>Array type</td>
</tr>
</tbody>
</table>

C++Script maps some types to standard types:

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

JScript has no types, all types are variants. BasicScript may have types (for example, `dim i as Integer`), or may have no types and even no variable declaration. In this case a variable will have Variant type.

Not all of these types can be assign-compatible. Like in Object Pascal, you can’t assign Extended or String to an Integer. Only one type - the Variant - can be assigned to all the types and can get value from any type.

Except the built-in types you can use the enumerated types defined in your application or in add-in modules (for example after adding the TfsGraphicsRTTI component you can use TPenMode, TFontStyles and other types).
### Classes

You cannot define a class inside the script, but you can use the external classes defined in add-in modules or your application. This is an example from the DEMOS\Main demo:

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

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

// there is no need to use all the parameters in event handlers
// because no type checking is performed here
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.
```

As you can see there is no difference between PascalScript and Delphi code. You can access any property (simple, indexed or default) or method. All the object’s published properties are accessible from the script by default. Public properties and methods need the implementation code - that’s why you can access it partially (for example, you cannot access the TForm.Print method or TForm.Canvas property because they are not implemented).

You can add your own classes - see "Scripting" chapter for details.
There is a rich set of standard functions which can be used in a script. 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
As you can see, some functions/procedure have default parameters. You can call it just like in Delphi:

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

You can connect own function/procedure to a script - see "Scripting" chapter for details.
Events

You can use event handlers in the script. Unlike the Delphi event handler, script event handlers are **not the methods of the object**. The following example shows how to connect an event handler to the TButton.OnClick event:

```pascal
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; // same as b.OnClick := 'ButtonClick'
  b.OnClick := nil; // clear the event
end.

There are some predefined events available in FS_iEvents unit:

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

See the "TfsFormsRTTI component", "TfsExtCtrlsRTTI component" and "TfsDBCtrlsRTTI component" chapters for a list of the available events.
**Enumerations and Sets**

FastScript supports enumerations. You can write in a script:

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

Sets are not supported. However, you can use set constants in the following way:

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

**Arrays**

FastScript supports all kind of arrays: static (one- and multi-dimesional), dynamic, variant arrays. There is an example of script that uses all array types:

```plaintext
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;
```
FastScript component palette

Delphi component palette
TfsScript component
TfsClassesRTTI component
TfsGraphicsRTTI component
TfsFormsRTTI component
TfsExtCtrlsRTTI component
TfsDialogsRTTI component
TfsDBRTTI component
TfsDBCtrlsRTTI component
TfsBDERTTI component
TfsADORTTI component
TfsIBXRTTI component

TfsChartRTTI component
**Delphi component palette**

After the FastScript installing the "FastScript" tab will be created in the Delphi / C++Builder. This tab contains the main FastScript components such as TfsScript, TfsClassesRTTI, etc.

**TfsScript component - the main scripting component**

This is a main scripting component.

**Properties:**

**SyntaxType: String;**

The type of the script language. By default four types of scripts are supported: "PascalScript", "C++Script", "BasicScript", "JScript". Warning! The property has the string type and it is easy to make a mistake in the syntax type indication. The value by default is “PascalScript”.

**Lines: TStrings;**

A script text. Contains strings of the script.

**Methods:**

**function Compile: Boolean;**

Compiles the source code. Source code must be placed in the TfsScript.Lines property before you call the Compile method.

**procedure Execute;**

Execute script after compiling.

**function Run: boolean;**

Compile and execute script. Returns true if compile was successful. This method is the analogue to the Compile + Execute.

**Examples of use:**

**Example1.**

Delphi/Kylix IDE. Loads script file MyTestScript.pas and execute it.
fsScript1.Lines.LoadFromFile('MyTestScript.pas');
if fsScript1.Compile then
  fsScript1.Execute
else
  ShowMessage('Script compilation error!');

**Example2.**
Delphi/Kylix IDE. Pressing the Button1 gives the strings from fsSyntaxMemo1 component to fsScript1.Lines and execute script.

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

**Example3.**
Delphi/Kylix IDE. Loads “C++Script” from MyTestScript.cpp file and execute it.

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

**Example4.**
C++Builder IDE. Loads “C++Script” from MyTestScript.cpp file and execute it.

```cpp
fsScript1->Lines->LoadFromFile('MyTestScript.cpp');
fsScript1->SyntaxType = "C++Script";
if (fsScript1->Compile())
  fsScript1->Execute();
else
  ShowMessage("Script compilation error!");
```
**TfsSyntaxMemo - script editor with syntax highlighting**

A special advanced TMemo is adapted for FastScript with support of C++ and Pascal syntax highlighting. *Available in FastScript for VCL only. It's a "bonus" component and is not supported at present time.*

**Properties:**

- **SyntaxType: TSyntaxType;**
  Type of syntax highlighting.
  Possible values:
  - stPascal - for Pascal,
  - stCpp - for C++,
  - stSQL - for SQL,
  - stText - a simple text (without highlighting).
  Default value *stPascal.*

- **Lines: TStrings;**
  The edit text.

- **ShowFooter: Boolean;**
  Shows the footer with a cursor position, etc.

- **ShowGutter: Boolean;**
  Shows the info in the left part of a text with bookmarks, current step position, etc.

- **BlockColor: TColor;**
  Background color of the selected text.

- **BlockFontColor: TColor;**
  Color of the selected text.

- **CommentAttr: TFont;**
  Attributes of the comment font.

- **KeywordAttr: TFont;**
  Attributes of keyword font.

- **StringAttr: TFont;**
  Attributes of the string values font.

- **TextAttr: TFont;**
Attributes of a simple text font.

**Modified:** Boolean;  
True if the text was modified.

**SelText:** String;  
Contains a selected text.

**Methods:**

**procedure CopyToClipboard:**  
Copies a selected text to the clipboard.

**procedure CutToClipboard:**  
Moves a selected text to the clipboard.

**procedure PasteFromClipboard:**  
Inserts a text from the clipboard to the cursor position.

**procedure SetPos(x, y: Integer):**  
Sets the cursor position in the text. Line and positions numbering begins from 0. See the **GetPos** method.

**function GetPos: TPoint:**  
Returns the current cursor position. See **SetPos**.

**procedure ShowMessage(s: String):**  
Shows a message in the footer.

**procedure Undo:**  
Cancels the last change.

**function Find(Text: String): boolean:**  
Searches a text from a current cursor position.

**function IsBookmark(Line : integer): integer:**  
Return the bookmark number for the line with the Line number. Returns -1 if the bookmark is not set. See **AddBookmark** method.

**procedure AddBookmark(Line, Number : integer):**  
Adds the bookmark number for the line Line with the number Number. Supports 10 bookmarks with numbers from 0 to 9. See **DeleteBookmark**, **GotoBookmark** methods.

**procedure DeleteBookmark(Number : integer):**  
Deletes the bookmark with the number Number. See **AddBookmark** method.
procedure GotoBookmark(Number : integer);
Sets the cursor position to the line with the bookmark with the number Number. See AddBookmark method.

procedure SetActiveLine(Line : Integer);
Sets the active line highlighting (for use with the debugger). Line is the active line number. The highlighting is disabled if Line is set to -1. See the GetActiveLine method.

function GetActiveLine: Integer;
Returns the active line number. If there is no active lines it returns -1. See SetActiveLine method.

Hot keys.

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cursor arrow</td>
<td>Cursor moving</td>
</tr>
<tr>
<td>PgUp, PgDn,</td>
<td>Page Up / Page Down</td>
</tr>
<tr>
<td>Ctrl+PgUp</td>
<td>Move to the begin of text</td>
</tr>
<tr>
<td>Ctrl+PgDn</td>
<td>Move to the end of text</td>
</tr>
<tr>
<td>Home</td>
<td>Move to the begin of line</td>
</tr>
<tr>
<td>End</td>
<td>Move to the end of line</td>
</tr>
<tr>
<td>Enter</td>
<td>Move to the next line</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete symbol at right or selected text</td>
</tr>
<tr>
<td>Backspace</td>
<td>Delete symbol at left</td>
</tr>
<tr>
<td>Ctrl+Y</td>
<td>Delete current line</td>
</tr>
<tr>
<td>Ctrl+Z</td>
<td>Undo last change</td>
</tr>
<tr>
<td>Shift+Стрелки курсора</td>
<td>Select the text block</td>
</tr>
<tr>
<td>Ctrl+A</td>
<td>Select all text</td>
</tr>
<tr>
<td>Ctrl+U</td>
<td>Unindent selected block</td>
</tr>
<tr>
<td>Ctrl+I</td>
<td>Indent selected block</td>
</tr>
<tr>
<td>Ctrl+C, Ctrl+Insert</td>
<td>Copy to clipboard</td>
</tr>
<tr>
<td>Ctrl+V, Shift+Insert</td>
<td>Paste from clipboard</td>
</tr>
<tr>
<td>Ctrl+X, Shift+Delete</td>
<td>Cut to clipboard</td>
</tr>
<tr>
<td>Ctrl+Shift+&lt;цифра&gt;</td>
<td>Set bookmark</td>
</tr>
<tr>
<td>Ctrl+&lt;цифра&gt;</td>
<td>Goto bookmark</td>
</tr>
<tr>
<td>Ctrl+F</td>
<td>Search text</td>
</tr>
<tr>
<td>F3</td>
<td>Continue search</td>
</tr>
</tbody>
</table>
**TfsTree - classes and functions tree-view**

Shows available classes and functions in a tree.
*It's a "bonus" component and is not supported at present time.*

**Properties:**

**property Script:** TfsScript;
TfsScript reference.

**property SyntaxMemo:** TfsSyntaxMemo; for VCL
**property SyntaxMemo:** TMemo; for CLX
Memo reference.

**property ShowClasses:** Boolean;
The function tree is shown.

**property ShowFunctions:** Boolean;
All the tree nodes are shown.

**property Expanded:** Boolean;
Expand all tree nodes.

**property ExpandLevel:** integer;
The level of the unfolded tree nodes. 2 by default.
**TfsClassesRTTI component**

Use this component if you want to get access to Classes.pas stuff in your application. This component allows you to access the following classes inside a script:

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

You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iClassesRTTI" unit to the "uses" clause.
**TfsGraphicsRTTI component**

Use this component if you want to get an access to Graphics.pas stuff in your application. This component allows you to access the following classes inside a script:

- **TFont**
- **TPen**
- **TBrush**
- **TCanvas**

```pascal
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**

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

- **TMetafile**
- **TMetafileCanvas**
- **TBitmap**

```pascal
property TBitmap.Canvas
```

- **type TFontStyles**
- **type TFontPitch**
- **type TPenStyle**
- **type TPenMode**
- **type TBrushStyle**

You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iGraphicsRTTI" unit to the "uses" clause.
**TfsFormsRTTI component**

Use this component if you want to get an access to StdCtrls.pas and Forms.pas stuff in your application. This component allows you to access the following classes inside a script:

- **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**
You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iFormsRTTI" unit to the "uses" clause.
**TfsExtCtrlsRTTI component**

Use this component if you want to get an access to ExtCtrls.pas stuff in your application. This component allows you to access the following classes inside a script:

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

TTreeView
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
You get an access to all the published properties of these classes and the access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iExtCtrlsRTTI" unit to the "uses" clause.
**TfsDialogsRTTI component**

Use this component if you want to get an access to Dialogs.pas stuff in your application. This component allows you to access the following classes inside a script:

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

Type definitions:

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

You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iDialogsRTTI" unit to the "uses" clause.
**TfsDBRTTI component**

Use this component if you want to get an access to DB.pas stuff in your application. This component allows you to access the following classes inside a script:

```pascal
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
T_BINARYFIELD
TBYTESFIELD
TVARBYTESFIELD
TBBCDFIELD

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.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
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

You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iDBRTTI" unit to the "uses" clause.
TfsDBCtrlsRTTI component

Use this component if you want to get an access to DBCtrls.pas stuff in your application. This component allows you to access the following classes inside a script:

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

You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iDBCtrlsRTTI" unit to the "uses" clause.
**TfsBDERTTI component**

Use this component if you want to get an access to BDE stuff in your application. This component allows you to access the following classes inside a script:

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

You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iBDERTTI" unit to the "uses" clause.
**TfsADORTTI component**

Use this component if you want to get an access to ADO stuff in your application. This component allows you to access the following classes inside a script:

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

You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iADORTTI" unit to the "uses" clause.
**TfsIBXRTTI component**

Use this component if you want to get an access to IBX stuff in your application. This component allows you to access the following classes inside a script:

TIBDatabase  
TIBTransaction  
TIBCUSTOMDATASET  
TIBTable  
TIBQuery  
procedure TIBQuery.ExecSQL  
TIBSTOREPS  
procedure TIBSTOREPS.ExecProc

You have an access to all the published properties of these classes and an access to some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iBXRRTTI" unit to the "uses" clause.
**TfsChartRTTI component**

Use this component if you want to get an access to TeeChart stuff in your application. This component allows you to access the following classes inside a script:

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
TCircleSeries
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 T_PanningMode
type TSeriesPointerStyle
type TMultiArea
type TMultiBar
type TBarStyle

You have an access to all the published properties of these classes and an access to
some public properties and methods.

Note: This is a "fake" component. It is needed only for automatic inclusion of the "FS_iChartRTTI" unit to the "uses" clause.
Scripting

The simplest example of scripting
Getting the list of the supported languages
Displaying the detail info about the syntax error
Debugging the the script
Adding a procedure to the script
Adding a function to the script
Adding a function with var and default parameters
Adding a function with class parameters
Adding a function which returns value of class type
Adding a constant to the script
Adding a variable to the script
Adding an object variable to the script
Adding a type to the script
Adding an enumeration to the script
Adding a set type to the script
Adding a class to the script
Implementing public properties and methods of the class
Implementing the event handler
Implementing non-standard event handler
Accessing script variables from the Delphi code
Calling a script function from the Delphi code
Calling a script function with var parameters
Calculation of the expressions
Saving and loading of the precompiled code
Script tutorials
**The simplest example of scripting**

Here is a sample code which demonstrates the easiest way of using FastScript. Just put the TfsScript, TfsPascal and TButton components onto your form and write the following code in the button.OnClick event:

```pascal
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;
```

- Clear the script. It is necessary if you use one component to run many scripts.
- Fill the Lines property by the script code;
- To use standard types and functions pass the fsGlobalUnit to the Parent property.
- Run the script using the PascalScript language. If compilation was successful, Run method returns True. Otherwise an error message is shown.

Another way to use TfsScript without fsGlobalUnit (for example, in multi-thread environment):

```pascal
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;
```

**Getting the list of the supported languages**

To get the list of the supported languages call the fsGetLanguageList(list: TStrings) procedure, defined in the FS_iTools unit.

```pascal
uses FS_iTools;

fsGetLanguageList(LangComboBox.Items);
```
**Displaying the detail info about the syntax error**

```pascal
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.ErrorMsg + ' at ' + fsScript1.ErrorPos;
      Exit;
    end
  else
    fsScript1.Execute;
end;
```

**Debugging the script**

Use OnRunLine. For example:

```pascal
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;
```

Examine the demo located in the DEMOS\Main folder.

**Adding a procedure to the script**

To add a procedure/function to a script, perform the following steps:
- Create a method handler - function of the TfsCallMethodEvent type.
- Call TfsScript.AddMethod method. The first parameter is a function syntax, the second is a link to the handler of TfsCallMethodEvent type.

```pascal
{ the function itself }
procedure TForm1.DelphiFunc(s: String; i: Integer);
begin
```
ShowMessage(s + ', ' + IntToStr(i));
end;

{ the method handler }
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;
  { make DelphiFunc procedure visible to a script }
  fsScript1.AddMethod('procedure DelphiFunc(s: String; i: Integer)',
    CallMethod);
  { compile the script }
  if fsScript1.Compile then
    fsScript1.Execute else  { execute if compilation was succesfull }
    ShowMessage(fsScript1.ErrorMsg); { show an error message }
end;

If you want to add several methods, you can do it using one method handler:

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

{ the method handler }
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const
MethodName: String;
var Params: Variant): Variant;
begin
  { dispatch the method call }
  if MethodName = 'DELPHIFUNC' then
    DelphiFunc(Params[0], Params[1])
  else if MethodName = 'DELPHIFUNC2' then
    DelphiFunc2(Params[0]);
end;

Adding a function to the script
The same as adding a procedure.

fsScript1.AddMethod('function DelphiFunc2(s: String): Boolean',
    CallMethod);
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String; var Params: Variant): Variant;
begin
Result := DelphiFunc(Params[0]);
end;

Adding a function with var and default parameters

You don't need to care about default parameters - they are substituted automatically by FastScript. Var parameters must be handled by you.

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

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

Adding a function with the class parameter

Since all the parameters are represented as the Variant array type, you need to convert them to objects.

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

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

Adding a function which returns a value of the class type

Since the values returned by the method handler is the array of the Variant type, you
need to convert the results of the TObject type to the Variant.

\[ \text{fsScript1.AddMethod('function MainForm: TForm', CallMethod);} \]

\{ the method handler \}
\begin{verbatim}
function TForm1.CallMethod(Instance: TObject; ClassType: TClass; const MethodName: String; 
var Params: Variant): Variant;
begin
  Result := Integer(Form1);
end;
\end{verbatim}

**Adding a constant to the script**

To add a constant to a script, call the TfsScript.AddConst method. The first parameter is the name of the constant, the second one is the type (it must be one of the standard types), the third one is a value.

\[ \text{fsScript1.AddConst('pi', 'Extended', 3.14159);} \]

**Adding a variable to the script**

To add a variable to a script, call the TfsScript.AddVariable method. It is similar to AddConst method, except that fact that you can change the value of a variable in a script. Note that the actual Delphi variable is not changed after the script execution.

\[ \text{fsScript1.AddVariable('i', 'Integer', i);} \]

**Adding an object to the script**

To add an object to a script, call the TfsScript.AddObject method. The first parameter is the name of the object, the second one is the object itself.

\[ \text{fsScript1.AddObject('Button1', Button1);} \]

If object has an unregistered type, you have to register it before calling AddObject:

\[ \text{fsScript1.AddClass(TForm1, 'TForm');} \]
\[ \text{fsScript1.AddObject('Form1', Form1);} \]

You can also use fsGlobalUnit.AddForm method to add a form or datamodule with all its child components:
fsGlobalUnit.AddForm(Form1);

In this case you don't need to register the form class by AddClass method. Now you can access a form element in the script:

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

**Adding a type to the script**

To add the own type to a script, call the TfsScript.AddType method. The first parameter is the name of the type, the second one is the one of the supported types:

```pascal
TfsVarType = (fvtInt, fvtBool, fvtFloat, fvtChar, fvtString, fvtClass, fvtArray, fvtVariant, fvtEnum);

fsScript1.AddType('TCursor', fvtInt);
```

**Adding an enumeration to the script**

To add an enumeration type to the script, call the TfsScript.AddEnum method. The first parameter is the name of the type, the second one is the type values separated by commas.

```pascal
fsScript1.AddEnum('TPrinterOrientation', 'poPortrait, poLandscape');
```
**Adding a set to the script**

To add a set type to a script, call the TfsScript.AddEnumSet method. The first parameter is the name of the type, the second one is the type values separated by commas.

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

**Adding a class to the script**

To add a class to a script, call the TfsScript.AddClass method. The first parameter is the class type, the second one is the name of the parent class.

```delphi
type
  TMyClass = class(TObject)
    ...
  end;
fsScript1.AddClass(TMyClass, 'TObject');
```

This will make all the published properties of this class available. If you want to make this class available for all the scripts, it is recommended to add this class to the fsGlobalUnit which is a global ancestor of all the scripts.

**Implementing public properties and methods of the class**

The AddClass method automatically adds all the published properties of the class. Public properties and methods require an extra work. The following example shows how to add a public method to a class. You need to create the method handler (function of type TfsCallMethod).

```delphi
begin
  ...
  { add new class inherited from TObject } with fsScript1.AddClass(TList, 'TObject') do begin
    { add public methods }
    AddMethod('function Add(Item: TObject): Integer', CallMethod);
    AddMethod('procedure Clear', CallMethod);
  end;
  ...
end;
```

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

  if MethodName = 'ADD' then
  { convert Variant parameter to Pointer type and pass it to Add method }
    TList(Instance).Add(Pointer(Integer(Params[0])))
  else if MethodName = 'CLEAR' then
    TList(Instance).Clear
end;

To implement a property you need to create a method handler and two types property handlers TfsGetValueEvent and 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;

Indexed and default properties are described by the method handler, the ordinary properties are handled by the Get/Set property handlers.

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;

{ method handler }
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
    TObject(Integer(Params[1])) :=
    TStrings(Instance).Objects[Params[0]] :=
    TStrings(Instance).Strings[Params[0]] := Params[1]
end;

{ property handler }
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;

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

Implementing the event handler

To add an event to the class, use the TfsClassVariable.AddEvent method. The first parameter is the event name, the second one is the event handler.

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

There are some predefined event handlers available in the FS_iEvents unit:

  TfsNotifyEvent
  TfsMouseEvent
  TfsMouseMoveEvent
  TfsKeyEvent
  TfsKeyPressEvent
  TfsCloseEvent
  TfsCloseQueryEvent
  TfsCanResizeEvent

See the the "TfsFormsRTTTI component", "TfsExtCtrlsRTTI component" and "TfsDBCtrlsRTTI component" chapters for the list of events available in script.

Implementing non-standard event handler

There are some predefined event handlers available in FS_iEvents unit:
TfsNotifyEvent
TfsMouseEvent
TfsMouseMoveEvent
TfsKeyEvent
TfsKeyPressEvent
TfsCloseEvent
TfsCloseQueryEvent
TfsCanResizeEvent

However, if you need to write your own event handler have a look at the following example:

{ example of two event handlers }

{ analogue of TNotifyEvent }
TfsNotifyEvent = class(TfsCustomEvent)
public
  procedure DoEvent(Sender: TObject);
  function GetMethod: Pointer; override;
end;

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

procedure TfsNotifyEvent.DoEvent(Sender: TObject);
begin
  { CallHandler is an internal method }
  CallHandler([Sender]);
end;

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

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

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

**Accessing script variables from the Delphi code**

To get/set the value of a script variables use TfsScript.Variables property.

```-delphi
test := fsScript1.Variables['i'];
fsScript1.Variables['i'] := 10;
```

**Calling a script function from the Delphi code**

To call a script function, use TfsScript.CallFunction method. The first parameter is the name of the called function, the second one is the function parameters.

```delphi
// call to 'function ScriptFunc(s: String; i: Integer)'
val := fsScript1.CallFunction('ScriptFunc', VarArrayOf(['hello', 1]));
```
**Calling a script function with var parameters**

The same as described above. Use TfsScript.CallFunction1 method if your procedure/function accepts var parameters:

```pascal
var
  Params: Variant;

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

**Calculation of the expressions**

If you want to calculate an expression (for example, 'i+1'), call the TfsScript.Evaluate method.

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

It is useful for debugging purposes.

**Saving and loading of the precompiled code**

Sometimes it is necessary to save compilation results and perform it later. You can do it with the help of the TfsScript.GetILCode and SetILCode methods. The below code compiles the source script and places the precompiled results to the stream:

```pascal
var
  s: TStream;

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

After this, you can restore the precompiled code from the stream and perform it:

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

**Using "uses" directive**
You can split large script to modules, like in Object Pascal:

File unit1.pas:

```pascal
uses 'unit2.pas';
begin
  Unit2Proc('Hello!');
end.
```

File unit2.pas:

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

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

As you can see, you should write module name with file extension in quotes. The code placed in `begin..end` of the included module will be executed when you run script (this is analogue of initialization in the Pascal).

In this example you cannot use unit1 from within unit2. This will cause circular reference and infinity loop when compiling such script. Such references are not allowed since FastScript does not have interface/implementation sections.

Using `#language` directive, you can write multi-language scripts. For example, one module may be written in PascalScript, another one - using C++Script:

File unit1.pas:

```pascal
uses 'unit2.pas';

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

File unit2.pas:

```pascal
#language C++Script

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

{
  ShowMessage("unit2 initialization, C++Script");
}
```
The #language directive must be the first line of the file. If this directive exists it overrides TfsScript.SyntaxType setting.

**Script tutorials**

Script tutorials are located in the DEMOS\Main\Samples folder. Compile the demo located in the DEMOS\Main folder and open the script samples in it.