TRbwParser

Pasdoc

July 22, 2005

Contents

Chapter 1

Unit RbwParser

1.1 Description

The `RbwParser` unit declares `TRbwParser`(1.3) along with associated classes and types.

1.2 Overview

`TFunctionRecord record` A `TFunctionRecord` is used to define a function that can be used in a TRbwParser. To use the function, you must first assign the fields of the TFunctionRecord and then call TRbwParser.Functions.Add.

`TVariables record` `TVariables` are used in the protected section of `TExpression`(1.3).

`ERbwParserError Class` `ERbwParserError` is raised if by `TRbwParser`(1.3) to report user errors and some programmer errors.

`TFunctionClass Class` `TFunctionClass` defines a function that can be called by `TRbwParser`(1.3).

`TFunctionStringList Class` `TFunctionStringList` maintains a sorted list of `TFunctionClass`(1.3)es.

`TConstant Class` `TConstant` defines storage for a constant value such as "Abc", 1, 2.5, or `True`.

`TCustomValue Class` `TCustomValue` is the abstract ancestor of `TCustomVariable`(1.3) and `TExpression`(1.3). It adds a `Name`(1.3) field to `TConstant`(1.3).

`TCustomVariable Class` `TCustomVariable` is the abstract ancestor of `TRealVariable`(1.3), `TIntegerVariable`(1.3), `TBooleanVariable`(1.3) and `TStringVariable`(1.3).

`TRealVariable Class` `TRealVariable` stores a double that may change from one execution of a `TExpression`(1.3) to the next.

`TIntegerVariable Class` `TIntegerVariable` stores a integer that may change from one execution of a `TExpression`(1.3) to the next.

`TBooleanVariable Class` `TBooleanVariable` stores a boolean that may change from one execution of a `TExpression`(1.3) to the next.

`TStringVariable Class` `TStringVariable` stores a string that may change from one execution of a `TExpression`(1.3) to the next.

`TExpression Class` `TExpression` is the compiled version of an Expression generated by the TRbwParser.`Compile`(1.3) method.

`TSpecialImplementor Class` `TSpecialImplementor` is used in conjunction with `SpecialImplentorList`(1.6) to create a descendent of `TExpression`(1.3) in TExpression.`New`(1.3).

`TSpecialImplentorList Class` `TSpecialImplentorList` is the type of the global variable `SpecialImplentorList`(1.6). It is used together with `TSpecialImplementor`(1.3) to create a descendent of `TExpression`(1.3) in TExpression.`New`(1.3).

`TRbwParser Class` `TRbwParser` compiles expression in strings via the `Compile`(1.3) method into `TExpression`(1.3) objects that it owns.

`TSelectExpression Class` `TSelectExpression` is used for "if" and "case" statements. It allows Evaluate to be faster and safer by only evaluating the arguments that will be used.

`Register`

`DataTypeToString`

`GenerateVariableName`

1.3 Classes, Interfaces, Objects and Records

TFunctionRecord record

Description

A `TFunctionRecord` is used to define a function that can be used in a TRbwParser. To use the function, you must first assign the fields of the TFunctionRecord and then call TRbwParser.Functions.Add.

```
TFunctionRecord = record
  InputDataTypes: array of TRbwDataType;
  OptionalArguments: integer;
  CanConvertToConstant: boolean;
  Name: string;
  Prototype: string;
  Hidden: boolean;
  case ResultType: TRbwDataType of
    rdtDouble: (RFunctionAddr: TRbwRealFunction);
    rdtInteger: (IFunctionAddr: TRbwIntegerFunction);
    rdtBoolean: (BFunctionAddr: TRbwBooleanFunction);
    rdtString: (SFunctionAddr: TRbwStringFunction);
```

end;

Fields

InputDataTypes

`InputDataTypes: array of TRbwDataType;`

`InputDataTypes` defines the type of data passed into the function assigned to RFunctionAddr, IFunctionAddr, BFunctionAddr, or SFunctionAddr. The data are passed in as pointers to variables of the correct type.

The length of `InputDataTypes` normally defines the maximum number of arguments that can be passed into the function. However, if OptionalArguments is less than 0, an unlimited number of arguments can be passed to the function and the types of all those arguments must match the type of the last member of InputDataTypes; In that case, the minimum number of required arguments is the length of InputDataTypes minus 1.

See also : `TRbwDataType`(1.5);

OptionalArguments

`OptionalArguments: integer;`

If `OptionalArguments` is greater than 0, some of the arguments passed to the function may be nil pointers. The nil pointers will always follow the non-nil pointers; no non-nil pointer may follow a nil pointer. The maximum number of nil pointers will be `OptionalArguments`.

If `OptionalArguments` is less than 0, the number of arguments passed to the function may be greater than the length of `InputDataTypes`(1.3). The minimum number of arguments is the length of `InputDataTypes`(1.3) minus 1. If more arguments are used, the types of those arguments must correspond to the value defined in the last member of InputDataTypes.

CanConvertToConstant

`CanConvertToConstant: boolean;`

`CanConvertToConstant` defines whether the result of a function may be considered a constant value if all of the values passed to the function in the values array are constants.

Normally `CanConvertToConstant` should be set to True but if the function makes reference to global variables that may change between one evaluation of the expression and the next, `CanConvertToConstant` should be set to False.

Pi is an example of a function for which CanConvertToConstant should be true.

`CanConvertToConstant` is used when optimizing compiled expressions.

Example:

```
var
  // ... others omitted in example.
   PiFunction : TFunctionRecord;
  SinFunction : TFunctionRecord;
```

Pi is a constant so CanConvertToConstant is true. Pi has no input variables so it will always be converted to a constant. When PiFunction is used to create a TFunctionClass, CanConvertToConstant will set the value of the TFunctionClass.AllowConversionToConstant property. That in turn will set the value of TExpression.AllowConversionToConstant. The same happens with RFunctionAddr and InputDataTypes.

_Pi is a function defined in the implementation

```
function _Pi(Values : array of pointer) : double;
begin
  result := Pi;
end;
```

Sin will always return the same value if it's input is a constant so CanConvertToConstant is set true. However, it will only be converted to a constant if all of it's input variables are constants.

_Sin is a function defined in the implementation

```
function _Sin(Values : array of pointer) : double;
begin
  result := Sin(PDouble(Values[0])^);
end;
```

```
constructor TFunctionStringList.Create;
begin
  inherited;
  CaseSensitive := False;
  Duplicates := dupError;
  Sorted := True;

  // ... others lines omitted in example.

   PiFunction.ResultType := rdtDouble;
  PiFunction.Name := 'Pi';
  SetLength(PiFunction.InputDataTypes, 0);
```

```
PiFunction.OptionalArguments := 0;
PiFunction.CanConvertToConstant := True;
PiFunction.RFunctionAddr := _Pi;
Add(PiFunction);

// ... others lines omitted in example.

 PiFunction.ResultType := rdtDouble;
PiFunction.Name := 'Pi';
SetLength(PiFunction.InputDataTypes, 0);
PiFunction.OptionalArguments := 0;
PiFunction.CanConvertToConstant := True;
PiFunction.RFunctionAddr := _Pi;
Add(PiFunction);

// ... others lines omitted in example.

end;
```

Name `Name: string;`

`Name` is the unique identifier of the function.

Prototype `Prototype: string;`

`Prototype` gives the names and arguments for a function as they should be shown on a GUI interface.

Hidden `Hidden: boolean;`

`Hidden` determines whether or not the function should be visible in a GUI interface.

Synonyms `Synonyms: array of string;`

`Synonyms` represents alternative valid spellings for the same function.

ResultType `ResultType: TRbwDataType`

`ResultType` defines the type of data returned by the function. It must correspond to the value assigned to the RFunctionAddr, IFunctionAddr, BFunctionAddr, or SFunctionAddr.

RFunctionAddr `RFunctionAddr`

`RFunctionAddr` is the address of the `TRbwRealFunction`(1.5) that is to be evaluated.

IFunctionAddr `IFunctionAddr`

`IFunctionAddr` is the address of the `TRbwIntegerFunction`(1.5) that is to be evaluated.

BFunctionAddr `BFunctionAddr`

`BFunctionAddr` is the address of the `TRbwBooleanFunction`(1.5) that is to be evaluated.

SFunctionAddr `SFunctionAddr`

`SFunctionAddr` is the address of the `TRbwStringFunction`(1.5) that is to be evaluated.

TVariables record

Description

`TVariables` are used in the protected section of `TExpression`(1.3).

```
TVariables = record
  Datum: Pointer;
  DataType: TRbwDataType;
end;
```

Fields

Datum `Datum: Pointer;`

`Datum` is a pointer to a value to be passed to a function.

DataType `DataType: TRbwDataType;`

`DataType` defines the type of data to which `Datum`(1.3) points.

ERbwParserError Class

Hierarchy

ERbwParserError > Exception

Description

`ERbwParserError` is raised if by `TRbwParser`(1.3) to report user errors and some programmer errors.

(Assertion failures represent programmer errors.)

Fields

ErrorType `implicit ErrorType: integer;`

`ErrorType`: integer;

Methods

CreateMode

Declaration `implicit constructor CreateMode(const Msg: string; const ErrorMode: integer);`

Description sets `ErrorType`(1.3) to ErrorMode.

TFunctionClass Class

Hierarchy

TFunctionClass > TObject

Description

`TFunctionClass` defines a function that can be called by `TRbwParser`(1.3).

Properties

AllowConversionToConstant `public property AllowConversionToConstant: boolean read GetAllowConversionToConstant write SetAllowConversionToConstant;`

`AllowConversionToConstant` defines whether the result of a function may be considered a constant value if all of the values passed to the function in the values array are constants.

Normally `AllowConversionToConstant` should be set to True but if the function makes reference to global variables that may change between one evaluation of the expression and the next, `AllowConversionToConstant` should be set to False.

Pi is an example of a function for which `AllowConversionToConstant` should be true.

`AllowConversionToConstant` is used when optimizing compiled expressions.

BFunctionAddr `public property BFunctionAddr: TRbwBooleanFunction read GetBFunctionAddr write SetBFunctionAddr;`

`BFunctionAddr` is the address of a `TRbwBooleanFunction`(1.5) assigned to the `TFunctionClass`(1.3). If `ResultType`(1.3) is of the wrong type, reading `BFunctionAddr` will raise an exception. Writing `BFunctionAddr` sets `ResultType`(1.3).

IFunctionAddr `public property IFunctionAddr: TRbwIntegerFunction read GetIFunctionAddr write SetIFunctionAddr;`

`IFunctionAddr` is the address of a `TRbwIntegerFunction`(1.5) assigned to the `TFunctionClass`(1.3). If `ResultType`(1.3) is of the wrong type,

reading `IFunctionAddr` will raise an exception. Writing `IFunctionAddr` sets `ResultType`(1.3).

InputDataCount

```
public property InputDataCount:  integer read
GetInputDataCount write SetInputDataCount;
```

`InputDataCount` normally defines the maximum number of arguments that can be passed into the function. However, if `OptionalArguments`(1.3) is less than 0, an unlimited number of arguments can be passed to the function and the types of all those arguments must match the type of the last member of `InputDataTypes`(1.3). In that case, the minimum number of required arguments is `InputDataCount` minus 1.

InputDataTypes

```
public property InputDataTypes[const Index:  integer]:
TRbwDataType read GetInputDataTypes write SetInputDataTypes;
```

`InputDataTypes` is used to define the data types passed into or returned by a function. Data are passed into a function as pointers to variables of the correct type. Results of the functions are values of the correct type rather than pointers.

Name

```
public property Name:  string read GetName write SetName;
```

`Name` defines the name of the function. The `Name` of each function and variable in a `TRbwParser`(1.3) must be unique.

Prototype

```
public property Prototype:  string read FPrototype write
FPrototype;
```

`Prototype` gives suggestions as to how the names and arguments for a function could be shown on a GUI interface. The "—" character is used in the prototype to classify functions. Portions of the prototype that appear before a "—" character is the classification to which the rest of the prototype belongs. Thus, the portions before the "—" could be used to populate a tree control.

OptionalArguments

```
public property OptionalArguments:  integer read
GetOptionalArguments write SetOptionalArguments;
```

If `OptionalArguments` is greater than 0, up to that number of nil pointers may be passed to the function when it is evaluated. All the nil pointers must follow all the non-nil pointers.

If `OptionalArguments` is less than 0, an unlimited number of arguments may be passed to the function. The data type of these pointers will correspond to the data type defined in the last member of `InputDataTypes`(1.3).

ResultType

```
public property ResultType:  TRbwDataType read
GetResultType;
```

`ResultType` is used to define the data type passed returned by a function. The results of the functions are values of the correct type rather than pointers.

Example:

Because OptionalArguments is less than 0, the CaseB, CaseI, CaseR, and CaseS functions can take an unlimited number of arguments.

```
      function _CaseBoolean(Values : ar-
ray of pointer) : boolean;
      begin
        result := PBoolean(Values[PInteger(Values[0])^])^;
      end;

      function _CaseInteger(Values : ar-
ray of pointer) : integer;
      begin
        result := PInteger(Values[PInteger(Values[0])^])^;
      end;

      function _CaseDouble(Values : ar-
ray of pointer) : double;
      begin
        result := PDouble(Values[PInteger(Values[0])^])^;
      end;

      function _CaseString(Values : ar-
ray of pointer) : String;
      begin
        result := PString(Values[PInteger(Values[0])^])^;
      end;

      var
        CaseBooleanFunction : TFunctionRecord;
        CaseIntegerFunction : TFunctionRecord;
        CaseDoubleFunction : TFunctionRecord;
        CaseStringFunction : TFunctionRecord;

      constructor TFunctionStringList.Create;
      begin
        inherited;
        CaseSensitive := False;
        Duplicates := dupError;
        Sorted := True;

        // ... others lines omitted in example.
```

```
 CaseBooleanFunction.ResultType := rdtBoolean;
CaseBooleanFunction.Name := 'CaseB';
SetLength(CaseBooleanFunction.InputDataTypes, 4);

CaseBooleanFunction.InputDataTypes[0] := rdtInteger;
CaseBooleanFunction.InputDataTypes[1] := rdtBoolean;
CaseBooleanFunction.InputDataTypes[2] := rdtBoolean;
CaseBooleanFunction.InputDataTypes[3] := rdtBoolean;
CaseBooleanFunction.CanConvertToConstant := True;
CaseBooleanFunction.OptionalArguments := -1;
CaseBooleanFunction.BFunctionAddr := _CaseBoolean;
Add(CaseBooleanFunction);

CaseIntegerFunction.ResultType := rdtInteger;
CaseIntegerFunction.Name := 'CaseI';
SetLength(CaseIntegerFunction.InputDataTypes, 4);

CaseIntegerFunction.InputDataTypes[0] := rdtInteger;
CaseIntegerFunction.InputDataTypes[1] := rdtInteger;
CaseIntegerFunction.InputDataTypes[2] := rdtInteger;
CaseIntegerFunction.InputDataTypes[3] := rdtInteger;
CaseIntegerFunction.OptionalArguments := -1;
CaseIntegerFunction.CanConvertToConstant := True;
CaseIntegerFunction.IFunctionAddr := _CaseInteger;
Add(CaseIntegerFunction);

CaseDoubleFunction.ResultType := rdtDouble;
CaseDoubleFunction.Name := 'CaseR';
SetLength(CaseDoubleFunction.InputDataTypes, 4);
CaseDoubleFunction.InputDataTypes[0] := rdtInteger;
CaseDoubleFunction.InputDataTypes[1] := rdtDouble;
CaseDoubleFunction.InputDataTypes[2] := rdtDouble;
CaseDoubleFunction.InputDataTypes[3] := rdtDouble;
CaseDoubleFunction.OptionalArguments := -1;

CaseDoubleFunction.CanConvertToConstant := True;
CaseDoubleFunction.RFunctionAddr := _CaseDouble;
Add(CaseDoubleFunction);

CaseStringFunction.ResultType := rdtString;
CaseStringFunction.Name := 'CaseS';
SetLength(CaseStringFunction.InputDataTypes, 4);
CaseStringFunction.InputDataTypes[0] := rdtInteger;
CaseStringFunction.InputDataTypes[1] := rdtString;
CaseStringFunction.InputDataTypes[2] := rdtString;
```

```
  CaseStringFunction.InputDataTypes[3] := rdtString;
  CaseStringFunction.OptionalArguments := -1;
  CaseStringFunction.CanConvertToConstant := True;
  CaseStringFunction.SFunctionAddr := _CaseString;
  Add(CaseStringFunction);
  // ... others lines omitted in example.
end;
```

RFunctionAddr `public property RFunctionAddr: TRbwRealFunction read GetRFunctionAddr write SetRFunctionAddr;`

`RFunctionAddr` is the address of a `TRbwRealFunction`(1.5) assigned to the `TFunctionClass`(1.3). If `ResultType`(1.3) is of the wrong type, reading `RFunctionAddr` will raise an exception. Writing `RFunctionAddr` sets `ResultType`(1.3).

SFunctionAddr `public property SFunctionAddr: TRbwStringFunction read GetSFunctionAddr write SetSFunctionAddr;`

`SFunctionAddr` is the address of a `TRbwStringFunction`(1.5) assigned to the `TFunctionClass`(1.3). If `ResultType`(1.3) is of the wrong type, reading `SFunctionAddr` will raise an exception. Writing `SFunctionAddr` sets `ResultType`(1.3).

Hidden `public property Hidden: boolean read GetHidden write SetHidden;`

`Hidden` has no effect. It is intended to be used to indicate whether or not the function specified by the `TFunctionClass`(1.3) should be visible to the user.

Fields

FunctionRecord `private FunctionRecord: TFunctionRecord;`

`FunctionRecord`: `TFunctionRecord`(1.3); `FunctionRecord` stores the data represented by many of the public properties.

FPrototype `private FPrototype: string;`

`FPrototype`: string; See `Prototype`(1.3).

Methods

GetAllowConversionToConstant

Declaration `private function GetAllowConversionToConstant: boolean;`

Description See `AllowConversionToConstant`(1.3).

GetBFunctionAddr

Declaration `private function GetBFunctionAddr: TrbwBooleanFunction;`

Description Gets value of BFunctionAddr property

GetHidden

Declaration `private function GetHidden: boolean;`

Description See `Hidden`(1.3).

GetIFunctionAddr

Declaration `private function GetIFunctionAddr: TrbwIntegerFunction;`

Description Gets value of IFunctionAddr property

GetInputDataCount

Declaration `private function GetInputDataCount: integer;`

Description See `InputDataCount`(1.3).

GetInputDataTypes

Declaration `private function GetInputDataTypes(const Index: integer): TRbwDataType;`

Description See `InputDataTypes`(1.3).

GetName

Declaration `private function GetName: string;`

Description See `Name`(1.3).

GetOptionalArguments

Declaration `private function GetOptionalArguments: integer;`

Description See `OptionalArguments`(1.3).

GetResultType

Declaration `private function GetResultType: TRbwDataType;`

Description See `ResultType`(1.3).

GetRFunctionAddr

Declaration `private function GetRFunctionAddr: TrbwRealFunction;`

Description Gets value of RFunctionAddr property

GetSFunctionAddr

Declaration `private function GetSFunctionAddr: TrbwStringFunction;`

Description Gets value of SFunctionAddr property

SetAllowConversionToConstant

Declaration `private procedure SetAllowConversionToConstant(const Value: boolean);`

Description See `AllowConversionToConstant`(1.3).

SetBFunctionAddr

Declaration `private procedure SetBFunctionAddr(const Value: TrbwBooleanFunction);`

Description Sets value of BFunctionAddr property

SetHidden

Declaration `private procedure SetHidden(const Value: boolean);`

Description See `Hidden`(1.3).

SetIFunctionAddr

Declaration `private procedure SetIFunctionAddr(const Value: TrbwIntegerFunction);`

Description Sets value of IFunctionAddr property

SetInputDataCount

Declaration `private procedure SetInputDataCount(const Value: integer);`

Description See `InputDataCount`(1.3).

SetInputDataTypes

Declaration `private procedure SetInputDataTypes(const Index: integer; const Value: TRbwDataType);`

Description See `InputDataTypes`(1.3).

SetName

Declaration `private procedure SetName(const Value:  string);`

Description See `Name`(1.3).

SetOptionalArguments

Declaration `private procedure SetOptionalArguments(const Value:  integer);`

Description See `OptionalArguments`(1.3).

SetRFunctionAddr

Declaration `private procedure SetRFunctionAddr(const Value:  TrbwRealFunction);`

Description Sets value of RFunctionAddr property

SetSFunctionAddr

Declaration `private procedure SetSFunctionAddr(const Value:  TrbwStringFunction);`

Description Sets value of SFunctionAddr property

Create

Declaration `public constructor Create;`

Description `Create` calls the inherited `Create` and sets `AllowConversionToConstant`(1.3) to true.

Users should generally not call `Create` directly but instead create a `TFunctionClass`(1.3) by calling TFunctionStringList.`Add`(1.3).

TFunctionStringList Class

Hierarchy

TFunctionStringList > TStringList

Description

`TFunctionStringList` maintains a sorted list of `TFunctionClass`(1.3)es.

In the functions below, except where otherwise noted, angles are expressed in radians.
The following functions are included:
AbsI(Value) returns the absolute value of Value. Value must be an integer. The value returned by AbsI will be an integer.
AbsR(Value) returns the absolute value of Value. Value can be either an integer or a real number. The value returned by AbsR will be a real number.
ArcCos(Value) returns the inverse cosine of Value. The return value is in the range from zero to Pi.

ArcCosh(Value) returns the inverse hyperbolic cosine of Value.
ArcSin(Value) returns the inverse sine of Value. The return value is in the range from -Pi/2 to +Pi/2.
ArcSinh(Value) returns the inverse hyperbolic sine of Value.
ArcTan2(Y, X) returns the inverse tangent of Y/X in the correct quadrant. The return value is in the range from -Pi to +Pi.
ArcTanh(Value) returns the inverse hyperbolic tangent of Value.
CaseB(Index, boolean_Result1, boolean_Result2, ...). CaseB uses Index to determine which of the boolean_Result1, boolean_Result2, ... arguments will be returned as a result. If Index equals 1, boolean_Result1 is returned; if Index equals 2, boolean_Result2 is returned; if Index equals 3, boolean_Result3 is returned; and so forth. Only "Index", constant expressions and the result that is returned will be evaluated. The types of boolean_Result1, boolean_Result2, ... must all be booleans. The type that is returned will be a boolean.
CaseI(Index, integer_Result1, integer_Result2, ...). CaseI uses Index to determine which of the integer_Result1, integer_Result2, ... arguments will be returned as a result. If Index equals 1, integer_Result1 is returned; if Index equals 2, integer_Result2 is returned; if Index equals 3, integer_Result3 is returned; and so forth. Only "Index", constant expressions and the result that is returned will be evaluated. The types of integer_Result1, integer_Result2, ... must all be integers. The type that is returned will be an integer.
CaseR(Index, real_Result1, real_Result2, ...). CaseR uses Index to determine which of the real_Result1, real_Result2, ... arguments will be returned as a result. If Index equals 1, real_Result1 is returned; if Index equals 2, real_Result2 is returned; if Index equals 3, real_Result3 is returned; and so forth. Only "Index", constant expressions and the result that is returned will be evaluated. The types of real_Result1, real_Result2, ... must all be real numbers. The type that is returned will be a real number.
CaseT(Index, text_Result1, text_Result2, ...). CaseT uses Index to determine which of the text_Result1, text_Result2, ... arguments will be returned as a result. If Index equals 1, text_Result1 is returned; if Index equals 2, text_Result2 is returned; if Index equals 3, text_Result3 is returned; and so forth. Only "Index", constant expressions and the result that is returned will be evaluated. The types of text_Result1, text_Result2, ... must all be character strings. The type that is returned will be a character string. CaseS is a synonym for CaseT.
Copy(text_Value, StartIndex, Count) returns a substring of text_Value starting at the character indicated by StartIndex and extending for either Count characters or until the end text_Value is reached whichever is smaller.
Cos(Value) returns the cosine of Value.
Cosh(Value) returns the hyperbolic cosine of Value.
DegToRad(Value) converts Value from degrees to radians. See also RadToDeg.
Distance(X1, Y1, X2, Y2) calculates the distance between points (X1,Y1) and (X2, Y2).
FactorialI(Value_Less_than_13) returns the factorial of Value_Less_than_13 as an integer.
FactorialR(Value_Less_than_171) returns the factorial of Value_Less_than_171 as a real number.
FloatToText(Value) converts the real number Value to its string representation. FloatToStr is a synonym for FloatToText.
Frac(Value) returns the fractional part of Value.
IfB(Boolean_Value, If_True_boolean_Result, If_False_boolean_Result). IfB uses Boolean_Value to determine whether If_True_boolean_Result or If_False_boolean_Result is returned as a result. If Boolean_Value is true, If_True_boolean_Result is returned; if Boolean_Value is false, If_False_boolean_Result is returned. Only "Boolean_Value", constant expressions and the result that is returned will be evaluated. The types of If_True_boolean_Result and If_False_boolean_Result must be boolean. The type that is returned will be a boolean.

IfI(Boolean_Value, If_True_integer_Result, If_False_integer_Result). IfI uses Boolean_Value to determine whether If_True_integer_Result or If_False_integer_Result is returned as a result. If Boolean_Value is true, If_True_integer_Result is returned; if Boolean_Value is false, If_False_integer_Result is returned. Only "Boolean_Value", constant expressions and the result that is returned will be evaluated. The types of If_True_integer_Result and If_False_integer_Result must be integers. The type that is returned will be an integer.
IfR(Boolean_Value, If_True_real_Result, If_False_real_Result). IfR uses Boolean_Value to determine whether If_True_real_Result or If_False_real_Result is returned as a result. If Boolean_Value is true, If_True_real_Result is returned; if Boolean_Value is false, If_False_real_Result is returned. Only "Boolean_Value", constant expressions and the result that is returned will be evaluated. The types of If_True_real_Result and If_False_real_Result must be real numbers. The type that is returned will be a real number.
IfT(Boolean_Value, If_True_text_Result, If_False_text_Result). IfT uses Boolean_Value to determine whether If_True_text_Result or If_False_text_Result is returned as a result. If Boolean_Value is true, If_True_text_Result is returned; if Boolean_Value is false, If_False_text_Result is returned. Only "Boolean_Value", constant expressions and the result that is returned will be evaluated. The types of If_True_text_Result and If_False_text_Result must be character strings. The type that is returned will be a character string. IfS is a synonym for IfT.
Interpolate(Position, Value1, Distance1, Value2, Distance2). Interpolate returns
(Position-Distance1)/(Distance2-Distance1)*(Value2-Value1)+Value1.
As its name implies, this is an interpolation between Value1 and Value2 based on where Position is between Distance1 and Distance2. See also `MultiInterpolate`(1.6).
IntPower(Base, Exponent) returns Base raised to the Exponent power. Base is a real number or integer. Exponent is an integer. See also Power.
IntToText(Value) converts the integer number Value to its string representation. IntToStr is a synonym for IntToText.
Length(text_Value) returns the number of characters in text_Value.
ln(Value) returns the natural log of Value.
log10(Value) returns the log to the base 10 of Value.
logN(Base, Value) returns the log to the base N of Value.
LowerCase(text_Value) returns text_Value with all its characters converted to lower case. See also: UpperCase.
MaxI(integer_Value1, integer_Value2, ...) returns whichever of its arguments is the largest. Its arguments must be integers. The result will be an integer.
MaxR(real_Value1, real_Value2, ...) returns whichever of its arguments is the largest. Its arguments must be either integers or real numbers. The result will be a real number.
MinI(integer_Value1, integer_Value2, ...) returns whichever of its arguments is the smallest. Its arguments must be integers. The result will be an integer.
MinR(real_Value1, real_Value2, ...) returns whichever of its arguments is the smallest. Its arguments must be either integers or real numbers. The result will be a real number.
Odd(Value) returns True if Value is an odd number. Otherwise it returns False. Value must be an integer.
Pi returns the ratio of the circumference of a circle to its diameter.
Pos(SubString, StringValue) returns the position of the first instance of SubString within StringValue. If Substring does not occur within StringValue, Pos returns 0.
PosEx(SubText, TextValue, Offset) returns the position of the first instance of SubText within TextValue that starts on or after Offset. If SubText does not occur within TextValue, on or after Offset, PosEx returns 0. If Offset equals one, PosEx is equivalent to Pos.

Power(Base, Exponent) returns Base raised to the Exponent power. Base and Exponent are real numbers or integers. See also IntPower.
RadToDeg(Value) converts Value from radians to degrees. See also DegToRad.
Round(Value) converts Value to the nearest integer. In the case of a number that is exactly halfway between two integers, it converts it to whichever one is even. See also Trunc.
Sin(Value) returns the sine of Value.
Sinh(Value) returns the hyperbolic sine of Value.
SqrI(integer_Value) returns integer_Value squared. Integer_Value must be an integer. The result of SqrI will be an integer.
SqrR(real_Value) returns real_Value squared. Real_Value can be either an integer or a real number. The result of SqrR will be a real number.
Sqrt(Value) returns the square root of Value.
TextToFloat(text_Value) converts text_Value to a real number. If text_Value can not be converted, TextToFloat causes an error. StrToFloat is a synonym for TextToFloat.
TextToFloatDef(text_Value, DefaultResult) converts text_Value to a real number. If text_Value can not be converted, DefaultResult is returned instead. StrToFloatDef is a synonym for TextToFloatDef.
TextToInt(text_Value) converts text_Value to an integer. If text_Value can not be converted, TextToInt causes an error. StrToInt is a synonym for TextToInt.
TextToIntDef(text_Value, DefaultResult) converts text_Value to an integer. If text_Value can not be converted, DefaultResult is returned instead. StrToIntDef is a synonym for TextToIntDef.
Tan(Value) returns the tangent of Value.
Tanh(Value) returns the hyperbolic tangent of Value.
Trunc(Value) truncates Value to an integer by rounding it towards zero. See also Round.
UpperCase(text_Value) returns text_Value with all its characters converted to upper case. See also LowerCase.

Properties

FunctionClass `public property FunctionClass[const Index: integer]: TFunctionClass read GetFunctionClass;`

`FunctionClass` returns the `TFunctionClass`(1.3) at position index.

Methods

GetFunctionClass

Declaration `private function GetFunctionClass(const Index: integer): TFunctionClass;`

Description See `FunctionClass`(1.3).

Add

Declaration `public function Add(const FunctionRecord: TFunctionRecord): Integer; reintroduce;`

Description `Add` creates a `TFunctionClass`(1.3) based on FunctionRecord, calls the TStringList.AddObject method using the function name and the `TFunctionClass`(1.3) and returns the `TFunctionClass`(1.3) that it created. Any `TFunctionClass`(1.3) created in this way is owned by the TFunctionStringList.

Clear

Declaration `public procedure Clear; override;`

Description `Clear` sets the number of items in the TFunctionStringList to 0 and Free's all the TFunctionClass it created.

Create

Declaration `public constructor Create;`

Description `Create` creates a TFunctionStringList and also creates a variety of `TFunctionClass`(1.3)'es.

Users should not generally call `Create` directly but instead use the `TFunctionStringList`(1.3) already created in TRbwParser.`Functions`(1.3).

Delete

Declaration `public procedure Delete(Index: Integer); override;`

Description `Delete` destroys the `TFunctionClass`(1.3) at the position Index.

Destroy

Declaration `public destructor Destroy; override;`

Description `Destroy` destroys the `TFunctionStringList`(1.3). Do not call `Destroy` directly. Call Free instead.

TConstant Class

Hierarchy

TConstant > TObject

Description

`TConstant` defines storage for a constant value such as "Abc", 1, 2.5, or `True`.

Properties

ResultString `protected property ResultString: string read FResultString write SetResultString;`

if `FResultType`(1.3) is rdtString, ResultString is the string that is stored.

ResultType `public property ResultType: TRbwDataType read FResultType;`

`ResultType` defines the type of data stored in the `TConstant`(1.3).

Fields

FResultString `private FResultString: string;`

`FResultString`: string; See `ResultString`(1.3). Ordinarily access to FResultString should be avoided.

FResult `protected FResult: Pointer;`

`FResult`: Pointer; `FResult` points to the address of the result.

FResultType `protected FResultType: TRbwDataType;`

`FResultType`: `TRbwDataType`(1.5); `FResultType` specifies what type of data is stored in FResult.

Methods

SetResultString

Declaration `private procedure SetResultString(const Value: string);`

Description See `ResultString`(1.3).

Create

Declaration `protected constructor Create(const DataType: TRbwDataType); overload;`

Description Create a TConstant and specify `FResultType`(1.3).

ValueToString

Declaration `protected function ValueToString: string;`

Description Convert the result to a string.

Create

Declaration `public constructor Create(Value: string); overload;`

Description `Create` creates a `TConstant` and sets `ResultType`(1.3).

Create

Declaration `public constructor Create(const Value: Boolean); overload;`

Description `Create` creates a `TConstant` and sets `ResultType`(1.3).

Create

Declaration `public constructor Create(const Value: integer); overload;`

Description `Create` creates a `TConstant` and sets `ResultType`(1.3).

Create

Declaration `public constructor Create(const Value: double); overload;`

Description `Create` creates a `TConstant` and sets `ResultType`(1.3).

Destroy

Declaration `public destructor Destroy; override;`

Description `Destroy` destroys the `TConstant`(1.3). Do not call `Destroy` directly. Call Free instead.

StringResult

Declaration `public function StringResult: string;`

Description `StringResult` returns the value stored in `TConstant` if that value is a string. Other wise it generates an error.

BooleanResult

Declaration `public function BooleanResult: boolean;`

Description `BooleanResult` returns the value stored in `TConstant` if that value is a boolean. Other wise it generates an error.

IntegerResult

Declaration `public function IntegerResult: Integer;`

Description `IntegerResult` returns the value stored in `TConstant` if that value is a integer. Other wise it generates an error.

DoubleResult

Declaration `public function DoubleResult: double;`

Description `DoubleResult` returns the value stored in `TConstant` if that value is a integer or double. Other wise it generates an error.

Decompile

Declaration `public function Decompile: string; virtual;`

Description `Decompile` converts the value stored in the `TConstant` to a string that can be compiled into an equivalent `TConstant`.

TCustomValue Class

Hierarchy

TCustomValue > `TConstant`(1.3) > TObject

Description

`TCustomValue` is the abstract ancestor of `TCustomVariable`(1.3) and `TExpression`(1.3). It adds a `Name`(1.3) field to `TConstant`(1.3).

Properties

Name `public property Name: string read FName;`

`Name` defines how the `TCustomValue`(1.3) should be identified. It is set via Create.

Fields

FName `private FName: string;`

`FName`: string; Always upper case

FUserName `private FUserName: string;`

`FUserName`: string; FUserName is the mixed-case version of `FName`(1.3). It is used in `Decompile`(1.3).

Methods

Create

Declaration `public constructor Create(const VariableName: string; const DataType: TRbwDataType);`

Description `Create` calls the inherited constructor and sets `Name`(1.3).

Decompile

Declaration `public function Decompile: string; override;`

Description `Decompile` returns the `Name`(1.3) of the `TCustomValue`(1.3).

TCustomVariable Class

Hierarchy

TCustomVariable > `TCustomValue`(1.3) > `TConstant`(1.3) > TObject

Description

TCustomVariable is the abstract ancestor of TRealVariable(1.3), TIntegerVariable(1.3), TBooleanVariable(1.3) and TStringVariable(1.3).

It validates the Name(1.3) in Create(1.3).

Methods

Create

Declaration `implicit constructor Create(const VariableName: string; const DataType: TRbwDataType);`

Description Create calls inherited create and then validates VariableName. Variable Name(1.3)s must start with a letter or underscore. The rest of the name must consist of letters underscores and digits.

TRealVariable Class

Hierarchy

TRealVariable > TCustomVariable(1.3) > TCustomValue(1.3) > TConstant(1.3) > TObject

Description

TRealVariable stores a double that may change from one execution of a TExpression(1.3) to the next.

Properties

Value `public property Value: double read GetValue write SetValue;`

Value is the double stored by TRealVariable(1.3).

Methods

GetValue

Declaration `private function GetValue: double;`

Description See Value(1.3).

SetValue

Declaration `private procedure SetValue(const Value: double);`

Description See Value(1.3).

Create

Declaration `public constructor Create(const VariableName: string);`

Description Create calls the inherited create.

TIntegerVariable Class

Hierarchy

TIntegerVariable > `TCustomVariable`(1.3) > `TCustomValue`(1.3) > `TConstant`(1.3) > TObject

Description

`TIntegerVariable` stores a integer that may change from one execution of a `TExpression`(1.3) to the next.

Properties

Value `public property Value: Integer read GetValue write SetValue;`

`Value` is the Integer stored by `TIntegerVariable`(1.3).

Methods

GetValue

Declaration `private function GetValue: Integer;`

Description See `Value`(1.3).

SetValue

Declaration `private procedure SetValue(const Value: Integer);`

Description See `Value`(1.3).

Create

Declaration `public constructor Create(const VariableName: string);`

Description `Create` calls the inherited create.

TBooleanVariable Class

Hierarchy

TBooleanVariable > `TCustomVariable`(1.3) > `TCustomValue`(1.3) > `TConstant`(1.3) > TObject

Description

`TBooleanVariable` stores a boolean that may change from one execution of a `TExpression`(1.3) to the next.

Properties

Value `public property Value: Boolean read GetValue write SetValue;`

`Value` is the Boolean stored by `TBooleanVariable`(1.3).

Methods

GetValue

Declaration `private function GetValue: Boolean;`

Description See `Value`(1.3).

SetValue

Declaration `private procedure SetValue(const Value: Boolean);`

Description See `Value`(1.3).

Create

Declaration `public constructor Create(const VariableName: string);`

Description `Create` calls the inherited create.

TStringVariable Class

Hierarchy

TStringVariable > `TCustomVariable`(1.3) > `TCustomValue`(1.3) > `TConstant`(1.3) > TObject

Description

`TStringVariable` stores a string that may change from one execution of a `TExpression`(1.3) to the next.

Properties

Value `public property Value: string read GetValue write SetValue;`
`Value` is the String stored by `TStringVariable`(1.3).

Methods

GetValue

Declaration `private function GetValue: string;`

Description See `Value`(1.3).

SetValue

Declaration `private procedure SetValue(const Value: string);`

Description See `Value`(1.3).

Create

Declaration `public constructor Create(const VariableName: string);`

Description `Create` calls the inherited create.

TExpression Class

Hierarchy

TExpression > `TCustomValue`(1.3) > `TConstant`(1.3) > TObject

Description

`TExpression` is the compiled version of an Expression generated by the TRbwParser.`Compile`(1.3) method.

It can be evaluated by using the Evaluate method and the result can then be read using the `BooleanResult`(1.3) method, `DoubleResult`(1.3) method, `IntegerResult`(1.3) method, or `StringResult`(1.3) method. The correct one to read can be determined from the `ResultType`(1.3) property. Every instance of `TExpression` is owned by the instance of `TRbwParser`(1.3) that compiled it.

Properties

AllowConversionToConstant

```
private property AllowConversionToConstant: boolean read
FAllowConversionToConstant write
SetAllowConversionToConstant;
```

`AllowConversionToConstant` defines whether the result of a function may be considered a constant value if all of the values passed to the function in the values array are constants.

Normally `AllowConversionToConstant` should be set to True but if the function makes reference to global variables that may change between one evaluation of the expression and the next, `AllowConversionToConstant` should be set to False.

Pi is an example of a function for which `AllowConversionToConstant` should be true.

`AllowConversionToConstant` is used when optimizing compiled expressions.

Variables

```
private property Variables[const Index: integer]:
TConstant read GetVariables write SetVariables;
```

`Variables` define the data passed to the function when `Evaluate`(1.3) is called. Any descendent of `TConstant`(1.3) may be assigned to `Variables`.

Tag

```
public property Tag: integer read FTag write FTag;
```

`Tag` has no predefined meaning. The Tag property is provided for the convenience of developers. It can be used for storing an additional integer value or it can be typecast to any 32-bit value such as a component reference or a pointer.

VariablesUsed

```
public property VariablesUsed:  TStringList read
GetVariablesUsed;
```

`VariablesUsed` is a list of all the variables used by the `TExpression`(1.3).

Fields

FOptionalArguments

```
private FOptionalArguments:  integer;
```

`FOptionalArguments`: integer; `FOptionalArguments` is the number of optional arguments in the function that will be evaluated.

FunctionAddr

```
private FunctionAddr:  Pointer;
```

`FunctionAddr`: Pointer; `FunctionAddr` is the address of the function used to evaluate the TExpression.

VariablesForFunction

```
private VariablesForFunction:  array of Pointer;
```

`VariablesForFunction`: array of Pointer; `VariablesForFunction` is the argument passed to the function used to evaluate the `TExpression`.

FAllowConversionToConstant

```
private FAllowConversionToConstant:  boolean;
```

`FAllowConversionToConstant`: boolean; See `AllowConversionToConstant`(1.3).

FTag

```
private FTag:  integer;
```

`FTag`: integer; See `Tag`(1.3)

StringVariableIndicies

```
private StringVariableIndicies:  array of integer;
```

`StringVariableIndicies`: array of integer; StringVariableIndicies indicates which members of `VariablesForFunction`(1.3) refer to strings.

StringVariableCount

```
private StringVariableCount:  integer;
```

`StringVariableCount`: integer; StringVariableCount is the number of strings in `VariablesForFunction`(1.3).

FVariablesUsed

```
private FVariablesUsed:  TStringList;
```

`FVariablesUsed`: TStringList; FVariablesUsed is used to hold the result of `VariablesUsed`(1.3).

FTopLevel

```
private FTopLevel:  boolean;
```

`FTopLevel`: boolean; FTopLevel is used in `Decompile`(1.3) to determine whether or not to include parenthesis around the outermost item.

ShouldEvaluate

```
protected ShouldEvaluate:  boolean;
```

`ShouldEvaluate`: boolean;

`ShouldEvaluate` is set to false if a `TExpression`(1.3) is equivalent to a `TConstant`(1.3). In such cases, the `TExpression`(1.3) is evaluated when

it is created and does not need to be reevaluated later. An example would be if the expression used to create the `TExpression`(1.3) was "1 + 1". This would be converted to "2".

Data

```
protected Data:  array of TVariables;
```

`Data`: array of `TVariables`(1.3); `Data` holds the arguments used to evaluate the `TExpression`(1.3).

Methods

Create

Declaration

```
private constructor Create(const VariableName:  string; const DataType:
TRbwDataType; const CanConvertToConstant:  boolean); overload;
```

Description Create a TExpression.

Create

Declaration

```
private constructor Create(const FunctionRecord:  TFunctionRecord);
overload;
```

Description Create a TExpression.

Create

Declaration

```
private constructor Create(const VariableName:  string; const DataType:
TRbwDataType); overload;
```

Description Create a TExpression.

Initalize

Declaration

```
private procedure Initalize(const FunctionAddress:  Pointer; const
DataTypes:  array of TRbwDataType; const OptionalArguments:  integer);
```

Description Initializes certain variables. Called by Create.

GetVariables

Declaration

```
private function GetVariables(const Index:  integer):  TConstant;
```

Description See `Variables`(1.3).

SetVariables

Declaration

```
private procedure SetVariables(const Index:  integer; const Value:
TConstant);
```

Description See `Variables`(1.3).

ConvertToConstant

Declaration `private function ConvertToConstant: TConstant;`

Description if the expression can be represented as a constant value, ConvertToConstant returns a `TConstant`(1.3) that represents that value. Otherwise, it returns nil.

ResetDataLength

Declaration `private procedure ResetDataLength(const Count: integer);`

Description If optional arguements are used, ResetDataLength ise used to resets the length of arrays to the correct length.

SetAllowConversionToConstant

Declaration `private procedure SetAllowConversionToConstant(const Value: boolean);`

Description See `AllowConversionToConstant`(1.3).

FillVariables

Declaration `private procedure FillVariables;`

Description `FillVariables` initializes `VariablesForFunction`(1.3) and `StringVariableIndicies`(1.3)

GetVariablesUsed

Declaration `private function GetVariablesUsed: TStringList;`

Description See `VariablesUsed`(1.3).

Create

Declaration `public constructor Create(const FunctionClass: TFunctionClass); overload; virtual;`

Description `Create` creates a `TExpression`(1.3) based on a `TFunctionClass`(1.3). The values of any variables used by the `TExpression`(1.3) must also be set before calling `Evaluate`(1.3).

Decompile

Declaration `public function Decompile: string; override;`

Description `Decompile` converts the value stored in the `TExpression`(1.3) to a string that can be compiled into an equivalent `TExpression`(1.3) together with its arguments.

Destroy

Declaration `public destructor Destroy; override;`

Description `Destroy` destroys the `TExpression`(1.3). Do not call Destroy directly. Call Free instead.

Evaluate

Declaration `public procedure Evaluate; virtual;`

Description `Evaluate` evaluates the expression and sets the result which may then be read using `BooleanResult`(1.3), `DoubleResult`(1.3), `IntegerResult`(1.3), or `StringResult`(1.3) depending on the `ResultType`(1.3).

Example:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  RbwParser1.Compile(Edit1.Text);
  RbwParser1.CurrentExpression.Evaluate;
  case RbwParser1.CurrentExpression.ResultType of
    rdtDouble:
       begin
        Label1.Caption := 'result: '
           + FloatToStr(RbwParser1.CurrentExpression.DoubleResult);
      end;
    rdtInteger:
       begin
        Label1.Caption := 'result: '
           + IntToStr(RbwParser1.CurrentExpression.IntegerResult);
      end;
    rdtBoolean:
       begin
        if RbwParser1.CurrentExpression.BooleanResult then
        begin
          Label1.Caption := 'result: ' + 'True';
        end
        else
        begin
          Label1.Caption := 'result: ' + 'False';
        end;
      end;
    rdtString:
       begin
        Label1.Caption := 'result: '
           + RbwParser1.CurrentExpression.StringResult;
      end;
```

```
    else Assert(False);
    end;
  end;
```

New

Declaration `public class function New(const FunctionClass: TFunctionClass): TExpression; virtual;`

Description `New` usually calls Create to create a `TExpression`(1.3) based on FunctionClass. However, in some cases, it creates a descendent of `TExpression`(1.3). Many of the descendents are declared in the implementation section of RBW_Parser.pas. Others can be generated via `SpecialImplentorList`(1.6). One such descendent is `TSelectExpression`(1.3) which overrides `Evaluate`(1.3).

UsesVariable

Declaration `public function UsesVariable(const Variable: TCustomVariable): boolean;`

Description `UsesVariable` returns True if Variable is used by the `TExpression`(1.3).

TSpecialImplementor Class

Hierarchy

TSpecialImplementor > TObject

Description

`TSpecialImplementor` is used in conjunction with `SpecialImplentorList`(1.6) to create a descendent of `TExpression`(1.3) in TExpression.`New`(1.3).

Fields

FunctionClass `public FunctionClass: TFunctionClass;`

`FunctionClass`: `TFunctionClass`(1.3); `FunctionClass` defines the interface of the expression.

Implementor `public Implementor: TExpressionClass;`

`Implementor`: `TExpressionClass`(1.5); `Implementor` is the class of TExpression that will be created.

TSpecialImplentorList Class

Hierarchy

TSpecialImplentorList > TObject

Description

`TSpecialImplentorList` is the type of the global variable `SpecialImplentorList`(1.6). It is used together with `TSpecialImplementor`(1.3) to create a descendent of `TExpression`(1.3) in TExpression.`New`(1.3).

Properties

Count `public property Count: integer read GetCount;`

`Count` is the number of `TSpecialImplementor`(1.3)s in the TSpecialImplentorList.

Capacity `public property Capacity: integer read GetCapacity write SetCapacity;`

`Capacity` is the number of `TSpecialImplementor`(1.3)s that the TSpecialImplentorList can hold without reallocating memory.

Setting `Capacity` to a value high enough to hold all the `TSpecialImplementor`(1.3)s that it will hold before `Add`(1.3)ing them, can speed up `Add`(1.3)ing the `TSpecialImplementor`(1.3)s.

Items `public property Items[const Index: integer]: TSpecialImplementor read GetItems write SetItems;`

`Items` provides access to the `TSpecialImplementor`(1.3)s held by the `TSpecialImplentorList`(1.3).

Fields

FList `private FList: TList;`

`FList`: TList; is used to store `TSpecialImplementor`(1.3)s.

Methods

GetCount

Declaration `private function GetCount: integer;`

Description See `Count`(1.3).

GetItems

Declaration `private function GetItems(const Index: integer): TSpecialImplementor;`

Description See `Items`(1.3).

SetItems

Declaration `private procedure SetItems(const Index: integer; const Value: TSpecialImplementor);`

Description See `Items`(1.3).

GetCapacity

Declaration `private function GetCapacity: integer;`

Description See `Capacity`(1.3).

SetCapacity

Declaration `private procedure SetCapacity(const Value: integer);`

Description See `Capacity`(1.3).

Create

Declaration `public constructor Create;`

Description `Create` creates an instance of TSpecialImplentorList.

Destroy

Declaration `public destructor Destroy; override;`

Description `Destroy` destroys a TSpecialImplentorList. Do not call `Destroy`. Call Free instead.

Add

Declaration `public function Add(const Item: TSpecialImplementor): Integer;`

Description `Add` adds a TSpecialImplementor to `Items`(1.3). The items that are added are not owned by the `TSpecialImplentorList`(1.3).

Clear

Declaration `public procedure Clear;`

Description `Clear` removes everything from `Items`(1.3) and sets the `Count`(1.3) and `Capacity`(1.3) to 0.

Delete

Declaration `public procedure Delete(const Index: Integer);`

Description `Delete` removes the `TSpecialImplementor`(1.3) at position Index from `Items`(1.3);

Remove

Declaration `public function Remove(const Item: TSpecialImplementor): Integer;`

Description `Remove` removes the `TSpecialImplementor`(1.3) indicated by Item from `Items`(1.3) and returns its former position. If it was not in `Items`(1.3) it returns -1.

TRbwParser Class

Hierarchy

TRbwParser > TComponent

Description

`TRbwParser` compiles expression in strings via the `Compile`(1.3) method into `TExpression`(1.3) objects that it owns.

It can also create `TRealVariable`(1.3)s, `TIntegerVariable`(1.3)s, `TBooleanVariable`(1.3)s, and `TStringVariable`(1.3)s that can be used in the `TExpression`(1.3) via the `CreateVariable`(1.3) method. Alternately, such variables created by another instance of TRbwParser can be used via the `RegisterVariable`(1.3) method.
All variables and expressions created by an instance of `TRbwParser` are owned by it.
If several TRbwParsers are used and variables owned by one are used in another one, `ClearExpressions`(1.3) and `ClearVariables`(1.3) should be called in all of them before any of them is destroyed.
See `TFunctionStringList`(1.3) and `OverloadedFunctionList`(1.6) for a list of the functions included by default.

Properties

CurrentExpression `public property CurrentExpression: TExpression read FCurrentExpression;`

`CurrentExpression` is the most recently `Compile`(1.3)d `TExpression`(1.3).

Expressions `public property Expressions[const Index: integer]: TExpression read GetExpressions;`

`Expressions` is an array of `TExpression`(1.3)s.

Functions `public property Functions: TFunctionStringList read FFunctions;`

`Functions` stores `TFunctionClass`(1.3)es in it's Objects property and manages the `TFunctionClass`(1.3)es used by TRbwParser.

Variables `public property Variables[const Index: integer]: TCustomValue read GetVariable;`

`Variables` is an array of variables (`TCustomValue`(1.3)s) used by the current `TRbwParser`(1.3).

Fields

FCurrentExpression `private FCurrentExpression: TExpression;`

`FCurrentExpression`: `TExpression`(1.3); See `CurrentExpression`(1.3).

FExpressions `private FExpressions: TStringList;`

`FExpressions`: TStringList; See `Expressions`(1.3).

FFunctions `private FFunctions: TFunctionStringList;`

`FFunctions`: `TFunctionStringList`(1.3); See `Functions`(1.3).

FVariables `private FVariables: TStringList;`

`FVariables`: TStringList; See `Variables`(1.3).

FOwnedVariables `private FOwnedVariables: TObjectList;`

`FOwnedVariables`: TObjectList; `FOwnedVariables` holds all the `TCustomVariable`(1.3)s owned by the `TRbwParser`(1.3).

Methods

GetExpressions

Declaration `private function GetExpressions(const Index: integer): TExpression;`

Description See `Expressions`(1.3).

GetVariable

Declaration `private function GetVariable(const Index: integer): TCustomValue;`

Description See `Variables`(1.3).

ClearExpressions

Declaration `public procedure ClearExpressions;`

Description `ClearExpressions` destroys all compiled `TExpression`(1.3)s in the `TRbwParser`(1.3).

If several `TRbwParser`(1.3)s are used and variables owned by one are used in another one, ClearExpressions and `ClearVariables`(1.3) should be called in all of them before any of them is destroyed.

ClearVariables

Declaration `public procedure ClearVariables;`

Description `ClearVariables` removes all variables from the list of variables used by `TRbwParser`(1.3) but does not free them (because they may be used by another `TRbwParser`(1.3)).

If several `TRbwParser`(1.3)s are used and variables owned by one are used in another one, `ClearExpressions`(1.3) and ClearVariables should be called in all of them before any of them is destroyed.

CreateVariable

Declaration `public function CreateVariable(const Name: string; const Value: boolean): TBooleanVariable; overload;`

Description `CreateVariable` creates a `TBooleanVariable`(1.3) owned by `TRbwParser`(1.3) and allows it to be used with the `TRbwParser`(1.3).

See also: `RegisterVariable`(1.3).

CreateVariable

Declaration `public function CreateVariable(const Name: string; const Value: integer): TIntegerVariable; overload;`

Description `CreateVariable` creates a `TIntegerVariable`(1.3) owned by `TRbwParser`(1.3) and allows it to be used with the `TRbwParser`(1.3).

See also: `RegisterVariable`(1.3).

CreateVariable

Declaration `public function CreateVariable(const Name: string; const Value: double): TRealVariable; overload;`

Description `CreateVariable` creates a `TRealVariable`(1.3) owned by `TRbwParser`(1.3) and allows it to be used with the `TRbwParser`(1.3).

See also: `RegisterVariable`(1.3).

Example:

```
 procedure TForm1.Button2Click(Sender: TObject);
var
  Index : integer;
  Variable : TRealVariable;
begin
  RbwParser1.CreateVariable('A', 2.5);
  RbwParser1.CreateVariable('B', 3.5);
  RbwParser1.Compile('A + B');
  RbwParser1.CurrentExpression.Evaluate;
  Label1.Caption := 'result: '
     + FloatToStr(RbwParser1.CurrentExpression.DoubleResult);
  // Label1.Caption is set to "result: 6".
   Index := RbwParser1.IndexOfVariable('a');
  // variable names are not case sensitive.
   Variable := RbwParser1.Variables[Index] as TRealVariable;
  Variable.Value := 6.5;

  RbwParser1.CurrentExpression.Evaluate;
  Label2.Caption := 'result: '
```

```
    + FloatToStr(RbwParser1.CurrentExpression.DoubleResult);
  // Label2.Caption is set to "result: 10".
   // As it stands, this event handler would cause
  // an error if it was called twice because
  // it would try to create variables named "A" and "B" twice.
   // Calling ClearVariables would prevent this if that is
  // what you really want to do.
end;
```

CreateVariable

Declaration `public function CreateVariable(const Name: string; const Value: string): TStringVariable; overload;`

Description `CreateVariable` creates a `TStringVariable`(1.3) owned by `TRbwParser`(1.3) and allows it to be used with the `TRbwParser`(1.3).

See also: `RegisterVariable`(1.3).

Compile

Declaration `public function Compile(var AString: string): integer;`

Description `Compile` searches for AString in `Expressions`(1.3). If it finds it, it sets `CurrentExpression`(1.3) to that `TExpression`(1.3) and returns it's position in `Expressions`(1.3). If it doesn't find it, Compile converts AString into a `TExpression`(1.3), sets `CurrentExpression`(1.3) to the `TExpression`(1.3) it created, adds the `TExpression`(1.3) to `Expressions`(1.3) and returns the position of the `TExpression`(1.3) in `Expressions`(1.3).

Expressions are owned by `TRbwParser`(1.3).

Create

Declaration `public constructor Create(AOwner: TComponent); override;`

Description `Create` creates an instance of `TRbwParser`(1.3);

DeleteExpression

Declaration `public procedure DeleteExpression(const Index: integer);`

Description `DeleteExpression` destroys the `TExpression`(1.3) at Index and deletes it from the list of `Expressions`(1.3).

Destroy

Declaration `public destructor Destroy; override;`

Description `Destroy` destroys the `TRbwParser`(1.3). Do not call Destroy directly. Call Free instead.

ExpressionCount

Declaration `public function ExpressionCount: integer;`

Description `ExpressionCount` returns the number of `TExpression`(1.3)s in `Expressions`(1.3).

IndexOfVariable

Declaration `public function IndexOfVariable(VariableName: string): integer;`

Description `IndexOfVariable` locates the `TCustomVariable`(1.3) whose name is VariableName in `Variables`(1.3) and returns it's position. If it is not in `Variables`(1.3) it returns -1.

RegisterVariable

Declaration `public procedure RegisterVariable(const Value: TCustomValue);`

Description `RegisterVariable` allows a `TCustomValue`(1.3) to be used by the `TRbwParser`(1.3) even though the `TCustomValue`(1.3) isn't owned by the `TRbwParser`(1.3).

RemoveExpression

Declaration `public procedure RemoveExpression(const Expression: TExpression);`

Description `RemoveExpression` deletes Expression from the list of `Expressions`(1.3) and destroys it.

RemoveVariable

Declaration `public procedure RemoveVariable(const Variable: TCustomVariable);`

Description If Variable is owned by or registered with `TRbwParser`(1.3), `RemoveVariable` removes it and, if it is owned by the current instance of `TRbwParser`(1.3), also destroys it. It also destroys any TExpression's in the current `TRbwParser`(1.3) that use Variable.

RenameVariable

Declaration `public procedure RenameVariable(var Index: integer; NewName: string);`

Description `RenameVariable` changes the name of the variable at position Index to a NewName and changes Index to the new position of the variable.

VariableCount

Declaration `public function VariableCount: integer;`

Description `VariableCount` is the number of variables in `Variables`(1.3).

TSelectExpression Class

Hierarchy

TSelectExpression > `TExpression`(1.3) > `TCustomValue`(1.3) > `TConstant`(1.3) > TObject

Description

`TSelectExpression` is used for "if" and "case" statements. It allows Evaluate to be faster and safer by only evaluating the arguments that will be used.

Methods

Evaluate

Declaration `public procedure Evaluate; override;`

Description `Evaluate` calls Evaluate for its first arguement and based on the result of that argument calls Evaluate for one of its other arguments and sets its own result to the result of that arguement.

1.4 Functions and Procedures

Register

Declaration `procedure Register;`

Description `Register` installs the component on the component palette.

DataTypeToString

Declaration `function DataTypeToString(const DataType: TRbwDataType): string;`

Description `DataTypeToString` converts a `TRbwDataType`(1.5) to a string. This function is primarily used in error messages.

GenerateVariableName

Declaration `function GenerateVariableName(const root: string): string;`

Description `GenerateVariableName` uses root to generate the name of a variable that could be accepted by `TRbwParser`(1.3) but does check whether a variable with that name already exists.

1.5 Types

TRbwRealFunction

Declaration `TRbwRealFunction = function(Values: array of pointer): double;`

Description `TRbwRealFunction` is the prototype for functions in `TRbwParser`(1.3) that return a real number. To use a `TRbwRealFunction`(1.5), assign it to a `RFunctionAddr`(1.3), assign the remainder of the fields in the `TFunctionRecord`(1.3) and then call `Add`(1.3) of `Functions`(1.3). Be sure that the `ResultType`(1.3) is rdtDouble when assigning a `TRbwRealFunction`(1.5). Failure to do so will lead to access violations or invalid results.

TRbwIntegerFunction

Declaration `TRbwIntegerFunction = function(Values: array of pointer): Integer;`

Description `TRbwIntegerFunction` is the prototype for functions in TRbwParser that return an integer. To use a TRbwIntegerFunction, assign it to the IFunctionAddr of a TFunctionRecord, assign the remainder of the fields in the TFunctionRecord and then call the Add function of TRbwParser.Functions. Be sure that the ResultType of the TFunctionRecord is rdtInteger when assigning a TRbwIntegerFunction. Failure to do so will lead to access violations or invalid results.

TRbwBooleanFunction

Declaration `TRbwBooleanFunction = function(Values: array of pointer): Boolean;`

Description `TRbwBooleanFunction` is the prototype for functions in TRbwParser that return a boolean. To use a TRbwBooleanFunction, assign it to the BFunctionAddr of a TFunctionRecord, assign the remainder of the fields in the TFunctionRecord and then call the Add function of TRbwParser.Functions. Be sure that the ResultType of the TFunctionRecord is rdtBoolean when assigning a TRbwBooleanFunction. Failure to do so will lead to access violations or invalid results.

TRbwStringFunction

Declaration `TRbwStringFunction = function(Values: array of pointer): string;`

Description `TRbwStringFunction` is the prototype for functions in TRbwParser that return a string. To use a TRbwStringFunction, assign it to the SFunctionAddr of a TFunctionRecord, assign the remainder of the fields in the TFunctionRecord and then call the Add function of TRbwParser.Functions. Be sure that the ResultType of the TFunctionRecord is rdtString when assigning a TRbwStringFunction. Failure to do so will lead to access violations or invalid results.

TRbwDataType

Declaration `TRbwDataType = (...);`

Description `TRbwDataType` is used to define the data type passed into or returned by a function. In the case of rdtString used to define the data passed into a function, the data are passed into the function as a PString. Data are passed into a function as pointers to variables of the correct type. Results of the functions are values of the correct type rather than pointers.

Values `rdtDouble`

`rdtInteger`

`rdtBoolean`

`rdtString`

TRbwDataTypes

Declaration `TRbwDataTypes = set of TRbwDataType;`

Description `TRbwDataTypes` is defined for convenience here but is not used by `TRbwParser`(1.3).

PFunctionRecord

Declaration `PFunctionRecord = ^TFunctionRecord;`

Description `PFunctionRecord` is a pointer to a `TFunctionRecord`(1.3).

PBoolean

Declaration `PBoolean = ^Boolean;`

Description `PBoolean` is a pointer to a boolean.

TExpressionClass

Declaration `TExpressionClass = class of TExpression;`

Description `TExpressionClass` is used in `TSpecialImplementor`(1.3) to create a descendent of `TExpression`(1.3).

1.6 Variables

OverloadedFunctionList

Declaration `OverloadedFunctionList: TObjectList;`

Description `OverloadedFunctionList` contains a series of `TFunctionClass`(1.3)es that define overloaded functions.

The following functions are included by default.

Abs(Value) returns the absolute value of Value. Value can be either an integer or a real number. The value returned by Abs will have the same type as Value.

Case(Index, Result1, Result2, ...). Case uses Index to determine which of the Result1, Result2, ... arguments will be returned as a result. If Index equals 1, Result1 is returned; if Index equals 2, Result2 is returned; if Index equals 3, Result3 is returned; and so forth. Only "Index", constant expressions and the result that is returned will be evaluated. The types of Result1, Result2, ... must all be the same but they can be of any type. The type that is returned will be the same as the type of Result1, Result2, ...

If(Boolean_Value, If_True_Result, If_False_Result). If uses Boolean_Value to determine whether If_True_Result or If_False_Result is returned as a result. If Boolean_Value is true, If_True_Result is returned; if Boolean_Value is false, If_False_Result is returned. Only "Boolean_Value", constant expressions and the result that is returned will be evaluated. The types of If_True_Result and If_False_Result must be the same but they can be of any type. The type that is returned will be the same as the type of If_True_Result and If_False_Result.

Max(Value1, Value2, ...) returns whichever of its arguments is the largest. Its arguments must be either integers or real numbers. The result will be a real number if any of the arguments is a real number. If all the arguments are integers, the result will be an integer.

Min(Value1, Value2, ...) returns whichever of its arguments is the smallest. Its arguments must be either integers or real numbers. The result will be a real number if any of the arguments is a real number. If all the arguments are integers, the result will be an integer.

MultiInterpolate(Position, Value1, Distance1, [Value2, Distance2,] ...). If Position is less than or equal to Distance1, MultiInterpolate returns Value1. If Position is greater than or equal to DistanceN, MultiInterpolate returns ValueN. If Position is between any two adjacent distances, linear interpolation between the associated values will be used to determine the value that will be returned. See also `Interpolate`(1.3).

Sqr(Value) returns Value Squared. Value can be either an integer or a real number. The result of Sqr will be an integer if Value is an integer. Otherwise it will be a real number.

SpecialImplentorList

Declaration `SpecialImplentorList: TSpecialImplentorList;`

Description See: `TSpecialImplentorList`(1.3).