Express Yourself

by Chris McNeil

Remember in school when your teachers explained how important math would be every day of your life? There is no way around it. From balancing your checkbook to calculating gas mileage, mathematical calculations just cannot be avoided. Computer applications are no different. Since we use computers to model real-world situations, calculations become a natural part of software development.

I recently developed a small application for a local office of an international company that deals in industrial equipment and materials. As distributors, they receive purchase orders for specific materials and equipment. In turn, they check with their suppliers for compatible parts at the lowest cost. Finally, they calculate their selling price using a set of formulae, factoring in a desired profit margin.

In many cases, the supplier will give a discounted purchase price. Based on business volume, these discounts will periodically fluctuate (sometimes several times per month).

Clearly, we have a situation where calculations are being performed on a dynamic set of formulae. This is the difficult task of application design. Should we hard-code and achieve a quick fix, or do we spend more time and build in flexibility. Too often, programmers hard-code solutions that will eventually require an application to be recompiled. I developed a more complete (read flexible) solution.

TExpression Component

`TExpression` is a non-visual component that provides a means of computing the result of an arithmetic expression, where the expression is represented as a string constant. Table 1 lists the operators and function calls supported by the expression parser. Notice that the expression can contain function calls. Each function accepts a single `Real` parameter and returns a `Real` result. User-defined functions can be added to the parser, as long as they conform to this specification.

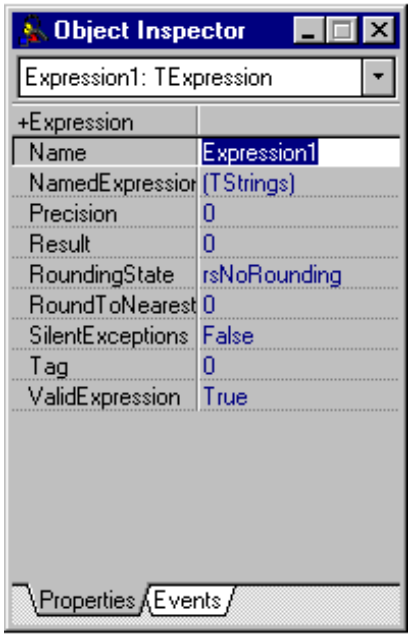
Operators	
Exponentiation	^, **
Multiplication	*, MULT
Division	/, DIV
Modulus	%, MOD
Addition	+
Subtraction	-
Functions	
ABS	Absolute Value
ARCTAN	ArcTangent
COS	Cosine
EXP	Exponential
FRAC	Fractional
INT	Integer
LN	Natural Logarithm
SIN	Sine
SQR	Square (Power of 2)
SQRT	Square Root

➤ *Table 1: TExpression operators and functions*

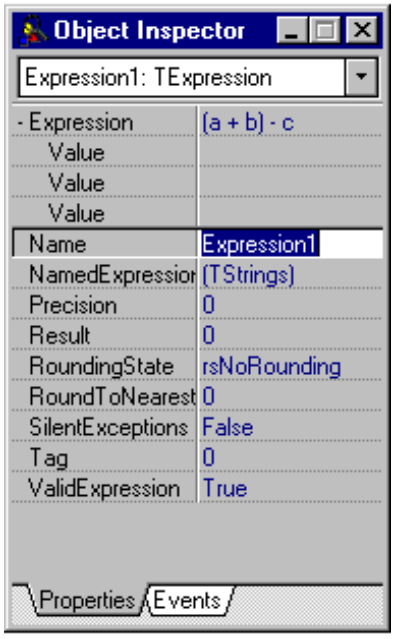
Expression	The expression to be evaluated.	
NamedExpressions	Expression storage mechanism.	
Precision	Number of decimal places to generate in the Result. Valid values are 0 and 2 through 8.	
Result	Result of the calculation. This is a read-only property.	
RoundingState	Standard and enhanced rounding capabilities:	
	rsStandardRounding	Standard, built-in rounding that would normally occur in Delphi calculations.
	rsAlwaysRoundDown	Directional rounding of the result toward 0; value of RoundToNearest used to determine how to round down.
	rsAlwaysRoundUp	Directional rounding of the result toward infinity; RoundToNearest used to determine how to round up.
RoundToNearest	Used for directional rounding. Valid values are 0 (no significant decimal places) and .01 through .99.	
SilentExceptions	Determines if exceptions are raised or silenced. However, the OnException event will always fire.	
ValidExpression	Determines when Expression is syntactically correct. This is a read-only property.	

➤ *Table 2: TExpression properties*

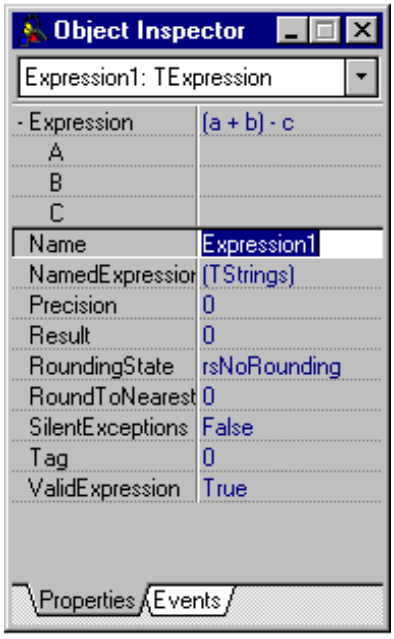
`TExpression` also provides a design time interface capable of expression validation, operand editing, standard and enhanced result rounding, plus handling and storage of named expressions.



➤ *Figure 1*



➤ *Figure 2*



➤ *Figure 3*

Terminology

A sample expression might be:

```
(5 / 9) * (F - 32)
```

This is the Fahrenheit to Celsius temperature conversion expression. It contains 4 operands: one is a variable (`F`) and three are numeric constants (5, 9 and 32). In addition, there are three operators (`/`, `*` and `-`).

Properties & Event Handlers

Table 2 lists the published properties of `TExpression`. In addition, `TExpression` publishes two event handlers: `OnCalculated` and `OnException`. `OnCalculated` fires just after a result has been calculated.

`OnException` fires when an internal exception occurs during `TExpression` processing and can be used to display an appropriate message to the user, or log the exception in an error log. The event will fire for all exceptions, even when `SilentExceptions` is `False`, and is of type `TExceptionEvent`.

Design-Time Interface

Figure 1 shows the Object Inspector with a `TExpression` component selected. `Expression` is a `string` property that supports sub-properties. Normally, sub-properties are supported by `TClassProperty` and `TSetProperty` descendants. However, by overriding the `TStringProperty` class (Listing 1) and including the `[paSubProperties]` attribute, this capability is possible. With this approach, the property editor is asked to provide its sub-properties for display in the Object Inspector. Ray Lischner's book *Secrets of Delphi 2* has a chapter called *Secrets of Property Editors*. It describes how to use surrogate components to "simulate" sub-properties. Ray's example involves splitting a `TDateTime` value into its constituent parts (`Month`, `Day`, `Year`, `Hours`, `Minutes`, `Seconds` and `Milliseconds`) using a single surrogate component. However, since the `Expression` property of `TExpression` can contain multiple, variable operands, we need to provide a way to access the operands with a slightly different approach.

The expression parser dissects the expression and generates a `TStringList` of `TOperand` components. Listing 2 defines `TOperand`, which holds the value of each variable operand. It's important to note that `TOperand` descends from `TComponent`. This is because the Delphi IDE can create an appropriate property editor for each published property of a component (check `GetComponentProperties` in the Help for more information).

Notice that the `TOperand` class has a single, published, property called `Value`. When each of the `TOperand` components (for a parsed expression) is passed to `GetComponentProperties`, a `TFloatProperty` editor is created.

The result is shown in Figure 2. Notice that each sub-property is listed in the Object Inspector with the name `Value`. Fortunately, the `TPropertyEditor` class, which is the ancestor of all property editors, provides the virtual `GetName` method. When overridden, this method is called by the Object Inspector whenever the name of a sub-property is required. The corrected version, now handled as a `TOperandProperty` (Listing 1), is shown in Figure 3.

This takes care of the sub-property list, but we still have a major problem. Typically, parent properties of sub-properties are read-only and simply display their class name in the Object Inspector, eg `Font` properties display as `(TFont)`. For `TExpression`, we need the `Expression` property to be editable, for obvious reasons. However, if the sub-property list is in an expanded state (`-`) when the `Expression` property is changed, the Object Inspector is not notified of the change. Consequently, the sub-property list is not properly updated. Figures 4 and 5 show this problem in action. The solution is to keep the `Expression` property

read-only when the sub-property list is expanded but editable when it's collapsed.

Toggling Property Attributes

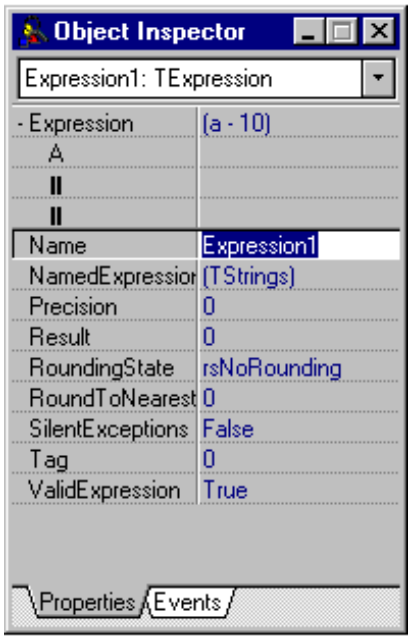
To reduce memory overhead in design mode, Delphi instantiates property editors only when they're needed. This means that just before a sub-property list is expanded, Delphi creates an appropriate property editor for each sub-property provided. It accomplishes this by calling the `GetProperties` method. To enforce the read-only state while expanded, a global `Boolean` variable (`bSubPropListExpanded`) was created and initialized to `False`. After all sub-property editors have been generated, `bSubPropListExpanded` is set to `True` and the `Modified` method of `TExpressionProperty` is called. This action causes the `GetAttributes` method to be called, which will encounter the new state of `bSubPropListExpanded`.

In doing so, the `[paReadOnly]` attribute gets included and the property becomes read-only.

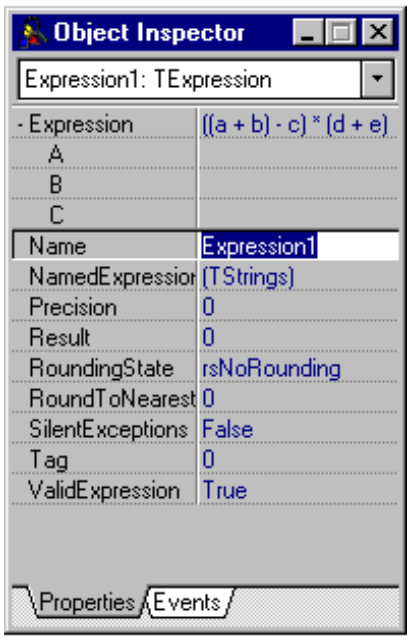
When the sub-property list is collapsed again, Delphi performs memory cleanup by destroying all non-essential property editors (for non-displayed sub-properties). Since we created our own `TOperandProperty` editor, we have access

➤ *Listing 1*

```
unit ExpReg;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes,
  Graphics, Controls, DsgnIntf, TypInfo, Dialogs, Forms,
  ExpComp, ExpEditr;
type
  TOperandProperty = class(TFloatProperty)
  public
    destructor Destroy; override;
    function   GetName: string; override;
    function   GetValue: string; override;
  end;
  TExpressionProperty = class(TStringProperty)
  protected
    procedure  ProcessOperand(Components: TStringList;
      Operand: TOperand);
  public
    function   GetAttributes: TPropertyAttributes; override;
    procedure  GetProperties(Proc: TGetPropEditProc);
      override;
  end;
  TResultProperty = class(TFloatProperty)
  public
    function   GetAttributes: TPropertyAttributes; override;
  end;
  TROEnumProperty = class(TEnumProperty)
  public
    function   GetAttributes: TPropertyAttributes; override;
    procedure  SetValue(const Value: string); override;
  end;
procedure Register;
var bSubPropListExpanded: Boolean;
implementation
procedure Register;
begin
  RegisterComponents('Samples', [TExpression]);
  RegisterPropertyEditor(TypeInfo(Extended), TOperand,
    '', TOperandProperty);
  RegisterPropertyEditor(TypeInfo(string), TExpression,
    'Expression', TExpressionProperty);
  ExpEditr.Register;    { NamedExpressions property editor }
  RegisterPropertyEditor(TypeInfo(Extended), TExpression,
    'Result', TResultProperty);
  RegisterPropertyEditor(TypeInfo(Boolean), TExpression,
    'ValidExpression', TROEnumProperty);
end;
destructor TOperandProperty.Destroy;
begin
  if (bSubPropListExpanded) then begin
    bSubPropListExpanded := False;
    Modified;
  end;
  inherited Destroy;
end;
function TOperandProperty.GetName: string;
begin
  Result := TOperand(GetComponent(0)).Name;
end;
function TOperandProperty.GetValue: string;
begin
  Result := inherited GetValue;
  if (Result = '-999999999') then
    Result := '';
end;
function TExpressionProperty.GetAttributes:
  TPropertyAttributes;
begin
  Result := inherited GetAttributes + [paSubProperties];
  if bSubPropListExpanded then
    Result := Result + [paReadOnly];
end;
procedure TExpressionProperty.ProcessOperand(
  Components: TStringList; Operand: TOperand);
var CompIdx: Integer;
begin
  { If Operand is not a System-Generated operand
    (Sys_Gen_Sym_), then generate an editor for its "Value"
    Property }
  if (Operand = nil) or (Pos('SYS_GEN_SYM_',
    UpperCase(Operand.Name)) <> 0) then exit;
  CompIdx := Components.IndexOf(UpperCase(Operand.Name));
  if (CompIdx = -1) then begin
    Components.AddObject(UpperCase(Operand.Name),
      TComponentList.Create);
    CompIdx := Components.Count - 1;
  end;
  TComponentList(Components.Objects[CompIdx]).Add(Operand);
end;
procedure TExpressionProperty.GetProperties(
  Proc: TGetPropEditProc);
var
  OpIdx:       Word;
  SelectIdx:   Word;
  CompIdx:     Word;
  OpCount:     Word;
  Operand:     TOperand;
  Expression:  TExpression;
  Components:  TStringList;
begin
  Components := TStringList.Create;
  { For each selected Expression (TExpression), extract all
   Operands within Expression and generate an appropriate
   Property Editor }
  try
    for SelectIdx := 0 to PropCount - 1 do begin
      Expression := TExpression(GetComponent(SelectIdx));
      OpCount := Expression.OperandCount;
      if (OpCount = 0) then
        continue;
      for OpIdx := 0 to OpCount - 1 do
        ProcessOperand(Components,
          Expression.GetOperandAt(OpIdx));
      end;
    if (Components.Count = 0) then begin
      Components.Free;
      Components := nil;
      exit;
    end;
    Components.Sorted := True;
    for CompIdx := 0 to Components.Count - 1 do begin
      GetComponentProperties(TComponentList(
        Components.Objects[CompIdx]), [tkFloat],
        Designer, Proc);
      TComponentList(
        Components.Objects[CompIdx]).Free;
    end;
    bSubPropListExpanded := True;
    Modified;
  finally
    Components.Free;
    Components := nil;
  end;
end;
function TResultProperty.GetAttributes: TPropertyAttributes;
begin
  Result := (inherited GetAttributes - [paMultiSelect]) +
   [paReadOnly];
end;
function TROEnumProperty.GetAttributes: TPropertyAttributes;
begin
  Result := (inherited GetAttributes - [paValueList]) +
    [paReadOnly];
end;
procedure TROEnumProperty.SetValue(const Value: string);
begin
  { paReadOnly keeps data from being keyed.  However, a
    double-click rotates through available values.
    Overriding this method ensures property IS NOT EDITABLE.}
end;
initialization
  bSubPropListExpanded := False;
end.
```



➤ *Figure 4*



➤ *Figure 5*

```
TOperand = class(TComponent)
  private
    FValue:    Extended;
    FOnChange: TNotifyEvent;
  protected
    procedure  SetValue(NewValue: Extended);
    procedure  Changed; virtual;
  public
    destructor Destroy; override;
    property   OnChange: TNotifyEvent
      read FOnChange write FOnChange;
  published
    property Value: Extended
      read FValue write SetValue;
end;
```

➤ *Listing 2*

to the `Destroy` method of each sub-property editor. It's in this method that we reset `bSubPropListExpanded` back to `False` and notify the `TExpressionProperty` (using the `Modified` method). Again, this forces a call to the `GetAttributes` method. This time, however, the `[paReadOnly]` attribute is not included and the `Expression` property is made editable again.

TOperand Value Changes

`TOperand` utilizes an `OnChange` event handler to broadcast value changes back to its owner `TExpression`. Upon `TOperand` instantiation, the event handler is set to the method `TExpression.RegisterOperandChange` using:

```
Operand.OnChange :=
  RegisterOperandChange;
```

`RegisterOperandChange` recalculates the result, if possible. As operand values are changed in the Object Inspector, the `OnChange` event fires, causing the expression result to be recalculated.

Run-Time Interface

The run-time interface is simple. First, establish the expression you wish to evaluate. Set the `Expression` property of `TExpression` to this value. Check the `ValidExpression` property to verify expression correctness. If `True`, the expression was successfully parsed. When the expression contains no variable operands, the `Result` property will be immediately calculated. However, if any variable operands are included, repeated calls to the `SetOperandValueByName` method must be made to establish values for these operands. For instance, if the following statement were executed:

```
Expression1.Expression :=
  '(5 / 9) * (F - 32)';
```

you would make a single call to `SetOperandValueByName` as follows:

```
Expression1.SetOperandValueByName(
  'f', 65);
```

(note that it's case-insensitive). As soon as the operand value is set, the `Result` property is available. In this case, a result of 18.33 is calculated.

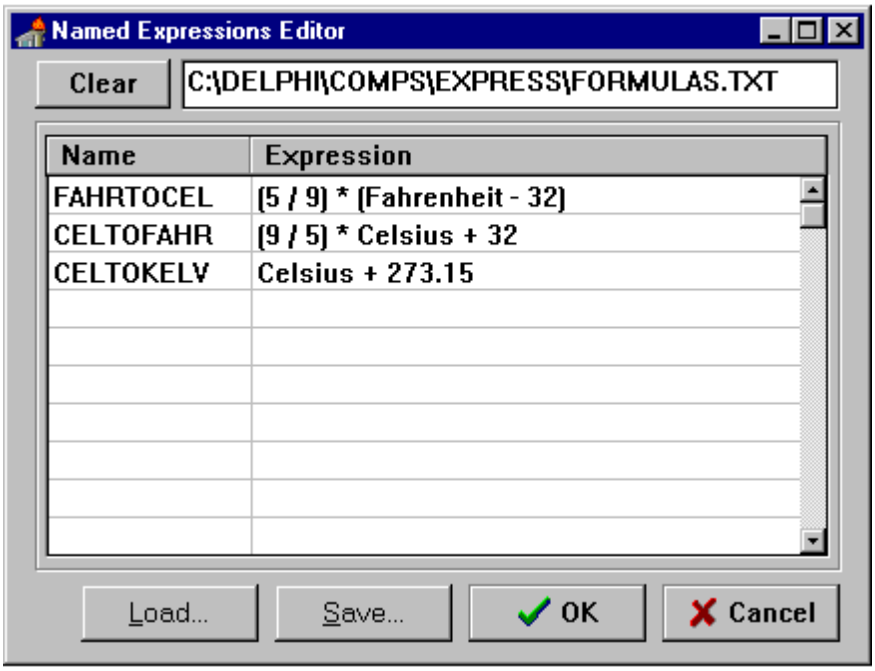
Named Expressions

The `NamedExpressions` property is a storage mechanism for often-used expressions. Normally, you'd expect expressions to be stored within database tables so that they could be manipulated outside of the application. This approach would shelter the application from simple changes to expression calculations. However, non-database applications require calculations too. It seems a bit extreme to carry several Mb of database support files (the BDE) just to allow for expression maintenance.

`NamedExpressions` is a `TStrings` property with its own property editor dialog (Figure 6). Notice the two-column string grid (`Name` and `Expression`). After keying all the necessary expression information, clicking `OK` will store each expression within the `NamedExpressions` property (and the application executable). By calling the public `GetNamedExpression` method, expressions can be retrieved for parsing. For instance, if the expressions from Figure 6 were stored, then the Fahrenheit to Celsius conversion expression would be accessed with:

```
Expression1.Expression :=
  Expression1.GetNamedExpression(
  'fahrtocel');
```

As before, this statement simply parses the expression. The result cannot be calculated until a call is made to `SetOperandValueByName`. If the expression were a simple calculation with no variable operands, the result would be calculated immediately.



➤ *Figure 6*

Obviously, expression storage is of limited use if expressions cannot be manipulated outside the Delphi IDE or the application executable. Therefore, the `TNamedExpressionsProperty` dialog has two additional buttons labelled `Load` and `Save`. These provide access to expressions stored in simple text files. When the `OK` button is clicked with a text file reference, as in Figure 6, the filename is stored in the `NamedExpressions` property rather than the expressions themselves. On `TExpression` instantiation, the stored filename is passed to the `LoadFromFile` method of the `TStrings` class. This loads the expressions into `NamedExpressions`. In this fashion, expressions can be maintained outside the application with no need for re-compilation.

Conclusion

The ability to manipulate arithmetic expressions outside of an application provides a very flexible technique for sheltering applications from constant programmer intervention. Using `TExpression`, subtle changes to calculations can be made easily with no need to recompile your applications. In addition, application users gain more control over the application and the information that it generates.

Chris McNeil is an independent developer specializing in Delphi component creation. He has been developing with Delphi since its inception. Prior to that, he wrote applications with Paradox for Windows and C++. He can be reached by email on Compuserve at 72734,2270